MetaCSV Documentation

Release 0.1.1

Michael Delgado

Sep 17, 2019

Contents

Contents:

CHAPTER 1

Installation

At the command line:

```
$ easy_install metacsv
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv metacsv
$ pip install metacsv
```

CHAPTER 2

Usage

To use MetaCSV in a project:

```
import metacsv
```

CHAPTER 3

metacsv package

3.1 Subpackages

3.1.1 metacsv.core package

Submodules

metacsv.core.containers module

metacsv.core.exceptions module

metacsv.core.internals module

Module contents

3.1.2 metacsv.io package

Submodules

metacsv.io.converters module

metacsv.io.parsers module

metacsv.io.to_csv module

metacsv.io.to_xarray module

metacsv.io.yaml_tools module

Module contents

3.1.3 metacsv.scripts package

Submodules

metacsv.scripts.convert module

metacsv.scripts.version module

Module contents

3.1.4 metacsv.testsuite package

Submodules

metacsv.testsuite.helpers module

metacsv.testsuite.test_metacsv module

Module contents

3.2 Module contents

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/delgadom/metacsv/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

MetaCSV could always use more documentation, whether as part of the official MetaCSV docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/delgadom/metacsv/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *metacsv* for local development.

1. Fork the *metacsv* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/metacsv.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv metacsv
$ cd metacsv/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 metacsv tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/delgadom/metacsv/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_metacsv
```

CHAPTER 5

Credits

5.1 Development Lead

- Michael Delgado <delgado.michaelt@gmail.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

Roadmap

- Make `coords` and `attrs` persistent across slicing operations (try `df['pop'].to_xarray()` from above example and watch it fail...)
- Improve hooks between `pandas` and `metacsv`:
  - update `coord` names on `df.index.names` assignment
  - update `coords` on stack/unstack
  - update `coords` on
- Handle attributes indexed by coord/variable names –> assign to coord/variable-specific `attrs`
- Let's start an issue tracker and get rid of this section!
- Should we rethink "special attributes," e.g. coords? Maybe these should have some special prefix like `_coords` when included in yaml headers to avoid confusion with other generic attributes...
- Allow special attributes (`coords`, `variables`) in `read_csv` call
- Allow external file headers
- Write tests
- Write documentation
- Maybe steal xarray's coordinate handling and save ourselves a whole lotta work?

6.1 Feature Requests

- Create syntax for `multi-csv` –> `Panel` or combining using filename regex
- Eventually? allow for on-disk manipulation of many/large files with dask/xarray
- Eventually? add xml, SQL, other structured syntax language conversions

CHAPTER 7

History

7.1 version dev

released Ongoing

Updated CHANGES.

7.2 version 0.0.1

released 2016-05-04

First release on PyPi.

`metacsv` - Tools for documentation-aware data reading, writing, and analysis

See the full documentation at ReadTheDocs

CHAPTER 8

Overview

MetaCSV provides tools to read in CSV data with a yaml-compliant header directly into a `pandas Series`, `DataFrame`, or `Panel` or an `xarray DataArray` or `Dataset`.

8.1 Data specification

Data can be specified using a yaml-formatted header, with the YAML *start-mark* string (`---`) above and the YAML *end-mark* string (`...`) below the yaml block. Only one yaml block is allowed. If the doc-separation string is not the first (non-whitespace) line in the file, all of the file's contents will be interpreted by the csv reader. The yaml data can have arbitrary complexity.

```
>>> import metacsv, numpy as np,
>>> import StringIO as io # import io for python 3
>>> doc = io.StringIO('''
---
author: A Person
date:   2000-12-31
variables:
    pop:
      name: Population
      unit: millions
    gdp:
      name: Product
      unit: 2005 $Bn
...
region,year,pop,gdp
USA,2010,309.3,13599.3
USA,2011,311.7,13817.0
CAN,2010,34.0,1240.0
CAN,2011,34.3,1276.7
''')
```

8.2 Using MetaCSV-formatted files in python

Read MetaCSV-formatted data into python using pandas-like syntax:

```
>>> df = metacsv.read_csv(doc, index_col=[0,1])
>>> df
<metacsv.core.containers.DataFrame (4, 2)>
               pop      gdp
region year
USA    2010  309.3  13599.3
       2011  311.7  13817.0
CAN    2010   34.0   1240.0
       2011   34.3   1276.7

Variables
    gdp:       OrderedDict([('name', 'Product'), ('unit', '2005 $Bn')])
    pop:       OrderedDict([('name', 'Population'), ('unit', 'millions')])
Attributes
    date:      2000-12-31
    author:    A Person
```

These properties can be transferred from one data container to another:

```
>>> s = metacsv.Series(np.random.random(6))
>>> s
<metacsv.core.containers.Series (6,)>
0    0.881924
1    0.556330
2    0.554700
3    0.221284
4    0.970801
5    0.946414
dtype: float64
>>> s.attrs = df.attrs
>>> s
<metacsv.core.containers.Series (6,)>
0    0.881924
1    0.556330
2    0.554700
3    0.221284
4    0.970801
5    0.946414
dtype: float64

Attributes
    date:      2000-12-31
    author:    A Person
```

All MetaCSV attributes, including the `attrs` Attribute object, can be copied, assigned to new objects, and deleted. Since these attributes are largely unstable across normal pandas data processing, it is recommended that attributes be copied before data work is attempted and then reassigned before IO conversions.

8.3 Exporting MetaCSV data to other formats

8.3.1 CSV

A MetaCSV `Series` or `DataFrame` can be written as a yaml-prefixed CSV using the same `to_csv` syntax as it's `pandas` counterpart:

```
>>> df.attrs['new attribute'] = 'changed in python!'
>>> df.to_csv('my_new_data.csv')
```

The resulting csv will include a yaml-formatted header with the original metadata updated to include attr['new attribute'].,

8.3.2 pandas

The coordinates and MetaCSV attributes can be easily stripped from a MetaCSV Container:

```
>>> df.to_pandas()
                pop       gdp
region year
USA    2010   309.3   13599.3
       2011   311.7   13817.0
CAN    2010    34.0    1240.0
       2011    34.3    1276.7
```

8.3.3 xarray/netCDF

xArray provides a pandas-like interface to operating on indexed `ndarray` data. It is modeled on the `netCDF` data storage format used frequently in climate science, but is useful for many applications with higher-order data.

```
>>> ds = df.to_xarray()
>>> ds
<xarray.Dataset>
Dimensions:  (region: 2, year: 2)
Coordinates:
  * region   (region) object 'USA' 'CAN'
  * year     (year) int64 2010 2011
Data variables:
    pop      (region, year) float64 309.3 311.7 34.0 34.3
    gdp      (region, year) float64 1.36e+04 1.382e+04 1.24e+03 1.277e+03
Attributes:
    date: 2000-12-31
    author: A Person
>>> ds.to_netcdf('my_netcdf_data.nc')
```

8.3.4 Pickling

Pickling works just like pandas.

```
>>> df.to_pickle('my_metacsv_pickle.pkl')
>>> metacsv.read_pickle('my_metacsv_pickle.pkl')
<metacsv.core.containers.DataFrame (4, 2)>
```

(continues on next page)

(continued from previous page)

```
               pop      gdp
region year
USA    2010  309.3  13599.3
       2011  311.7  13817.0
CAN    2010   34.0   1240.0
       2011   34.3   1276.7

Variables
    gdp:       OrderedDict([('name', 'Product'), ('unit', '2005 $Bn')])
    pop:       OrderedDict([('name', 'Population'), ('unit', 'millions')])
Attributes
    date:      2000-12-31
    author:    A Person
```

8.3.5 Others

Currently, MetaCSV only supports conversion to CSV and to netCDF through the `xarray` module. However, feel free to suggest additional features and to contribute your own!

8.4 Conversion to other types on the fly

Special conversion utilities allow you to convert any metacsv, pandas, or xarray container or a CSV filepath into any other type in this group.

All of these conversion utilities are also methods on metacsv containers.

- to_csv

`to_csv` allows you to write any container or csv file to a metacsv-formatted csv file. Keyword arguments `attrs`, `coords`, and `variables` will be attached to the data before it is written. Any conflicts in these attributes will be updated with the arguments to this function

```
>>> import pandas as pd, numpy as np, xarray as xr, metacsv
>>> df = pd.DataFrame(np.random.random((3,4)), columns=list('abcd'))
>>> df
          a         b         c         d
0  0.558083  0.665184  0.226173  0.339905
1  0.541712  0.835804  0.326078  0.179103
2  0.332869  0.435573  0.904612  0.823884

>>> metacsv.to_csv(df, 'mycsv.csv', attrs={'author': 'my name', 'date': '2015-12-31'})
>>>
>>> df2 = metacsv.read_csv('mycsv.csv', index_col=[0])
>>> df2
<metacsv.core.containers.DataFrame (3, 4)>
          a         b         c         d
0  0.558083  0.665184  0.226173  0.339905
1  0.541712  0.835804  0.326078  0.179103
2  0.332869  0.435573  0.904612  0.823884

Attributes
    date:      2015-12-31
    author:    my name
```

(continues on next page)

(continued from previous page)

```
>>> metacsv.to_csv(df2, 'mycsv.csv', attrs={'author': 'new name'})
>>>
>>> metacsv.read_csv('mycsv.csv', index_col=[0])
<metacsv.core.containers.DataFrame (3, 4)>
          a         b         c         d
0  0.558083  0.665184  0.226173  0.339905
1  0.541712  0.835804  0.326078  0.179103
2  0.332869  0.435573  0.904612  0.823884

Attributes
    date:      2015-12-31
    author:    new name
```

- to_header

`to_header` allows you to write the special attributes directly to a metacsv-formatted header file. The special attributes may be individually specified or taken from a metacsv container. The `header_file` argument to both `read_csv` and `to_csv` allow the creation of special header files which allow you to separate the metacsv-formatted header from the data if desired.

For example, say you have a table to read into pandas

```
>>> import metacsv, pandas as pd
>>> pd.DataFrame(
    [['x',1,2,3],['y',4,5,6],['z',7,8,9]], columns=['index','a','b','c']).to_csv(
↪'mycsv.csv', index=None)
>>> metacsv.read_csv('mycsv.csv')
<metacsv.core.containers.DataFrame (3, 4)>
  index  a  b  c
0     x  1  2  3
1     y  4  5  6
2     z  7  8  9
```

A separate header file can be created and used which can then be read in with the data:

```
>>> metacsv.to_header('mycsv.header', attrs={'author': 'me'}, coords='index')
>>> metacsv.read_csv('mycsv.csv', header_file='mycsv.header')
<metacsv.core.containers.DataFrame (3, 3)>
       a  b  c
index
x      1  2  3
y      4  5  6
z      7  8  9

Coordinates
  * index      (index) object x, y, z
Attributes
    author:         me
```

- to_xarray

`to_xarray` returns any container or csv file as an xarray container. Table data (CSV files and DataFrames) will create `xarray.Dataset` objects, while Series objects will create `xarray.DataArray` objects. Keyword arguments `attrs`, `coords`, and `variables` will be attached to the data before it is written. Any conflicts in these attributes will be updated with the arguments to this function.

- to_dataarray

`to_dataarray` returns any container or csv file as an `xarray.DataArray`. Table data (CSV files and DataFrames) will be stacked, with columns re-arranged as new `xarray.Coordinates`. Keyword arguments `attrs`, `coords`, and `variables` will be attached to the data before it is written. Any conflicts in these attributes will be updated with the arguments to this function.

- to_dataset

`to_dataarray` returns any container or csv file as an `xarray.DataArray`. Table data (CSV files and DataFrames) will be stacked, with columns re-arranged as new `xarray.Coordinates`. Keyword arguments `attrs`, `coords`, and `variables` will be attached to the data before it is written. Any conflicts in these attributes will be updated with the arguments to this function.

- to_pandas

`to_pandas` strips special attributes and returns an ordinary `Series` or `DataFrame` object.

- to_netcdf

`to_netcdf` first converts a container or csv file to an `xarray.Dataset` using the `to_dataset` function, then writes the dataset to file with the `xarray` `ds.to_netcdf` method.

```
>>> metacsv.to_netcdf('mycsv.csv', 'mycsv.nc', header_file='mycsv.header')
>>> import xarray as xr
>>> xr.open_dataset('mycsv.nc')
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) |S1 'x' 'y' 'z'
Data variables:
    a        (index) int64 1 4 7
    b        (index) int64 2 5 8
    c        (index) int64 3 6 9
Attributes:
    author: me
```

8.5 Special attributes

The `coords` and `variables` attributes are keywords and are not simply passed to the MetaCSV object's `attrs` attribute.

8.5.1 Variables

Variables are attributes which apply to speicific columns or data variables. In MetaCSV containers, variables are displayed as a separate set of attributes. On conversion to `xarray`, these attributes are assigned to variable-specific `attrs`:

```
>>> ds = df.to_xarray()
>>> ds
<xarray.Dataset>
Dimensions:  (index: 4)
Coordinates:
  * index    (index) int64 0 1 2 3
Data variables:
    region   (index) object 'USA' 'USA' 'CAN' 'CAN'
    year     (index) int64 2010 2011 2010 2011
```

(continues on next page)

(continued from previous page)

```
    pop      (index) float64 309.3 311.7 34.0 34.3
    gdp      (index) float64 1.36e+04 1.382e+04 1.24e+03 1.277e+03
Attributes:
    date: 2000-12-31
    author: A Person

>>> ds.pop
<xarray.DataArray 'pop' (index: 4)>
array([ 309.3,  311.7,   34. ,   34.3])
Coordinates:
  * index    (index) int64 0 1 2 3
Attributes:
    name: Population
    unit: millions
```

Note that at present, variables are not persistent across slicing operations.

parse_vars

Variables have a special argument to `read_csv`: `parse_vars` allows parsing of one-line variable definitions in the format `var:  description [unit]`:

```
>>> doc = io.StringIO('''
---
author: A Person
date:   2000-12-31
variables:
    pop: Population [millions]
    gdp: Product [2005 $Bn]
...
region,year,pop,gdp
USA,2010,309.3,13599.3
USA,2011,311.7,13817.0
CAN,2010,34.0,1240.0
CAN,2011,34.3,1276.7
''')

>>> metacsv.read_csv(doc, index_col=0, parse_vars=True)
<metacsv.core.containers.DataFrame (4, 3)>
        year    pop      gdp
region
USA     2010  309.3  13599.3
USA     2011  311.7  13817.0
CAN     2010   34.0   1240.0
CAN     2011   34.3   1276.7

Variables
    gdp:       {u'description': 'Product', u'unit': '2005 $Bn'}
    pop:       {u'description': 'Population', u'unit': 'millions'}
Attributes
    date:      2000-12-31
    author:    A Person
```

8.5.2 Coordinates

The conceptual foundation of coordinates is taken from `xarray`, where data is treated as an ndarray rather than a table. If you plan to only work with the pandas-like features of `metacsv`, you do not really need coordinates.

That said, specifying the `coords` attribute in a csv results in automatic index handling:

```
>>> doc = io.StringIO('''
---
author: A Person
date:   2000-12-31
variables:
    pop:
      name: Population
      unit: millions
    gdp:
      name: Product
      unit: 2005 $Bn
coords:
    - region
    - year
...
region,year,pop,gdp
USA,2010,309.3,13599.3
USA,2011,311.7,13817.0
CAN,2010,34.0,1240.0
CAN,2011,34.3,1276.7
''')

>>> df = metacsv.read_csv(doc)
>>> df
<metacsv.core.containers.DataFrame (4, 2)>
               pop      gdp
region year
USA    2010  309.3  13599.3
       2011  311.7  13817.0
CAN    2010   34.0   1240.0
       2011   34.3   1276.7

Coordinates
  * region     (region) object CAN, USA
  * year       (year) int64 2010, 2011
Variables
    gdp:       OrderedDict([('name', 'Product'), ('unit', '2005 $Bn')])
    pop:       OrderedDict([('name', 'Population'), ('unit', 'millions')])
Attributes
    date:      2000-12-31
    author:    A Person
```

Coordinates become especially useful, however, when moving to `xarray` objects or `netCDF` files. The `DataFrame` above will have no trouble, as `region` and `year` are orthoganal:

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (region: 2, year: 2)
Coordinates:
  * region   (region) object 'USA' 'CAN'
  * year     (year) int64 2010 2011
```

(continues on next page)

(continued from previous page)

```
Data variables:
    pop      (region, year) float64 309.3 311.7 34.0 34.3
    gdp      (region, year) float64 1.36e+04 1.382e+04 1.24e+03 1.277e+03
Attributes:
    date: 2000-12-31
    author: A Person
```

This becomes more complicated when columns in the index are not independent and cannot be thought of as orthogonal. In this case, you can specify `coords` as a dict-like attribute either in the CSV header or as an argument to the conversion method:

```
doc = io.StringIO('''
---
coords:
    region:
    regname: 'region'
    continent: 'region'
    year:
...
region,regname,continent,year,pop,gdp
USA,United States,North America,2010,309.3,13599.3
USA,United States,North America,2011,311.7,13817.0
CAN,Canada,North America,2010,34.0,1240.0
CAN,Canada,North America,2011,34.3,1276.7
''')

>>> metacsv.to_xarray(doc)
<xarray.Dataset>
Dimensions:    (region: 2, year: 2)
Coordinates:
  * region     (region) object 'USA' 'CAN'
  * year       (year) int64 2010 2011
    regname    (region) object 'United States' 'Canada'
    continent  (region) object 'North America' 'North America'
Data variables:
    pop        (region, year) float64 309.3 311.7 34.0 34.3
    gdp        (region, year) float64 1.36e+04 1.382e+04 1.24e+03 1.277e+03
```

Note that the resulting `Dataset` is not indexed by the cartesian product of all four coordinates, but only by the base coordinates, indicated by the `*`. Without first setting the `coords` attribute this way, the resulting data would have `NaN` values corresponding to `(USA, Canada)` and `(CAN, United States)`.

CHAPTER 9

TODO

- Allow automatic coersion of `xarray.Dataset` and `xarray.DataArray` objects to MetaCSV containers.
- Extend metacsv functionality to `Panel` objects
- Make `coords` and `attrs` persistent across slicing operations (try `df['pop'].to_xarray()` from above example and watch it fail...)
- Improve hooks between `pandas` and `metacsv`:
  - update `coord` names on `df.index.names` assignment
  - update `coords` on stack/unstack
  - update `coords` on
- Improve parser to automatically strip trailing commas and other excel relics
- Enable `read_csv(engine='C')...` this currently does not work.
- Handle attributes indexed by coord/variable names –> assign to coord/variable-specific `attrs`
- Let's start an issue tracker and get rid of this section!
- Should we rethink "special attribute," naming e.g. coords? Maybe these should have some special prefix like `_coords` when included in yaml headers to avoid confusion with other generic attributes...
- Allow attribute assertions (e.g. `version='>1.6.0'`) in `read_csv` call
- Improve test coverage
- Improve documentation & build readthedocs page

CHAPTER 10

Feature Requests

- Create syntax for `multi-csv` –> `Panel` or combining using filename regex
- Eventually? allow for on-disk manipulation of many/large files with dask/xarray
- Eventually? add xml, SQL, other structured syntax language conversions

CHAPTER 11

Indices and tables

- genindex
- modindex
- search