xarray-extras Documentation

Release 0.3.0

xarray-extras Developers

2019-01-02

Contents

1 Features **3**

2 Command-line tools **5**

3 Index **7**

3.1 Installation . . . 7
3.2 What's New . . . 8
3.3 csv . . . 9
3.4 cumulatives . . . 9
3.5 interpolate . . . 10
3.6 numba_extras . . . 12
3.7 recursive_diff . . . 12
3.8 sort . . . 15
3.9 stack . . . 16
3.10 testing . . . 16
3.11 ncdiff . . . 16
3.12 Usage . . . 16

4 Credits **19**

5 License **21**

Python Module Index **23**

This module offers several extensions to xarray, which could not be included into the main module because they fall into one or more of the following categories:

- They're too experimental
- They're too niche
- They introduce major new dependencies (e.g. numba or a C compiler)
- They would be better done by doing major rework on multiple packages, and then one would need to wait for said changes to reach a stable release of each package - *in the right order*.

The API of xarray-extras is unstable by definition, as features will be progressively migrated upwards towards xarray, dask, numpy, pandas, etc.

CHAPTER 1

Features

csv Multi-threaded CSV writer, much faster than `pandas.DataFrame.to_csv()`, with full support for dask and dask distributed.

cumulatives Advanced cumulative sum/productory/mean functions

interpolate dask-optimized n-dimensional spline interpolation

numba_extras Additions to numba

recursive_diff Recursively compare nested Python objects, with numpy/pandas/xarray support and tolerance for numerical comparisons

sort Advanced sort/take functions

stack Tools for stacking/unstacking dimensions

testing Tools for unit tests

CHAPTER 2

Command-line tools

bin/ncdiff.rst Compare two NetCDF files or recursively find all NetCDF files within two paths and compare them

CHAPTER 3

Index

3.1 Installation

3.1.1 Required dependencies

- Python 3.5 or later
- scipy
- xarray
- dask
- numba
- C compiler (only if building from sources)

3.1.2 Deployment

- With pip: **`pip install xarray-extras`**
- With anaconda: **`conda install -c conda-forge xarray-extras`**

3.1.3 Testing

To run the test suite after installing xarray_extras, first install (via pypi or conda)

- py.test: Simple unit testing library

and run `py.test --pyargs xarray_extras`.

3.2 What's New

3.2.1 v0.3.0 (2018-12-13)

- Changed license to Apache 2.0
- Increased minimum dask version to 0.19
- Increased minimum pandas version to 0.21
- New function `proper_unstack()`
- New functions `recursive_diff()` and `xarray_extras.testing.recursive_eq()`
- New command-line tool *ncdiff*
- Increased minimum xarray version to 0.10.1
- Increased minimum pytest version to 3.6
- Blacklisted Python 3.7 conda-forge builds in CI tests

3.2.2 v0.2.2 (2018-07-24)

- Fixed segmentation faults in `to_csv()`
- Added conda-forge travis build
- Blacklisted dask-0.18.2 because of regression in argtopk(split_every=2)

3.2.3 v0.2.1 (2018-07-22)

- Added parameter nogil=True to `to_csv()`, which will switch to a C-accelerated implementation instead of pandas to_csv (albeit with caveats). Fixed deadlock in to_csv as well as compatibility with dask distributed. Pandas code (when using nogil=False) is not wrapped by a subprocess anymore, which means it won't be able to use more than 1 CPU (but compression can run in pipeline). to_csv has lost the ability to write to a buffer - only file paths are supported now.
- AppVeyor integration

3.2.4 v0.2.0 (2018-07-15)

- New function `xarray_extras.csv.to_csv()`
- Speed up interpolation for k=2 and k=3
- CI: Rigorous tracking of minimum dependency versions
- CI: Explicit support for Python 3.7

3.2.5 v0.1.0 (2018-05-19)

Initial release.

3.3 csv

Multi-threaded CSV writer, much faster than `pandas.DataFrame.to_csv()`, with full support for dask and dask distributed.

`xarray_extras.csv.`**`to_csv`**(*x*, *path*, *, *nogil=True*, ***kwargs*)

Print DataArray to CSV.

When x has numpy backend, this function is functionally equivalent to (but much) faster than):

```
x.to_pandas().to_csv(path_or_buf, **kwargs)
```

When x has dask backend, this function returns a dask delayed object which will write to the disk only when its .compute() method is invoked.

Formatting and optional compression are parallelised across all available CPUs, using one dask task per chunk on the first dimension. Chunks on other dimensions will be merged ahead of computation.

Parameters

- **`x`** – xarray.DataArray with one or two dimensions
- **`path`** (*str*) – Output file path
- **`nogil`** (*bool*) – If True, use accelerated C implementation. Several kwargs won't be processed correctly (see limitations below). If False, use pandas to_csv method (slow, and does not release the GIL). nogil=True exclusively supports float and integer values dtypes (but the coords can be anything). In case of incompatible dtype, nogil is automatically switched to False.
- **`kwargs`** – Passed verbatim to `pandas.DataFrame.to_csv()` or `pandas.Series.to_csv()`

Limitations

- Fancy URIs are not (yet) supported.
- compression='zip' is not supported. All other compression methods (gzip, bz2, xz) are supported.
- When running with nogil=True, the following parameters are ignored: columns, quoting, quotechar, doublequote, escapechar, chunksize, decimal

Distributed

This function supports dask distributed, with the caveat that all workers must write to the same shared mountpoint and that the shared filesystem must strictly guarantee **close-open coherency**, meaning that one must be able to call write() and then close() on a file descriptor from one host and then immediately afterwards open() from another host and see the output from the first host. Note that, for performance reasons, most network filesystems do not enable this feature by default.

Alternatively, one may write to local mountpoints and then manually collect and concatenate the partial outputs.

3.4 cumulatives

Advanced cumulative sum/productory/mean functions

`xarray_extras.cumulatives.`**`cummean`**(*x*, *dim*, *skipna=None*)

$$y_i = mean(x_0, x_1, ...x_i)$$

Parameters

- **x** – any xarray object
- **dim** (*str*) – dimension along which to calculate the mean
- **skipna** (*bool*) – If True, skip missing values (as marked by NaN). By default, only skips missing values for float dtypes; other dtypes either do not have a sentinel missing value (int) or skipna=True has not been implemented (object, datetime64 or timedelta64).

xarray_extras.cumulatives.**compound_sum**(*x*, *c*, *xdim*, *cdim*)

Compound sum on arbitrary points of x along dim.

Parameters

- **x** – Any xarray object containing the data to be compounded
- **c** (*xarray.DataArray*) – array where every row contains elements of x.coords[xdim] and is used to build a point of the output. The cells in the row are matched against x.coords[dim] and perform a sum. If different rows of c require different amounts of points from x, they must be padded on the right with NaN, NaT, or '' (respectively for numbers, datetimes, and strings).
- **xdim** (*str*) – dimension of x to acquire data from. The coord associated to it must be monotonic ascending.
- **cdim** (*str*) – dimension of c that represent the vector of points to be compounded for every point of dim

Returns DataArray with all dims from x and c, except xdim and cdim, and the same dtype as x.

example:

```
>>> x = xarray.DataArray(
>>>     [10, 20, 30],
>>>     dims=['x'], coords={'x': ['foo', 'bar', 'baz']})
>>> c = xarray.DataArray(
>>>     [['foo', 'baz', None],
>>>      ['bar', 'baz', 'baz']],
>>>      dims=['y', 'c'], coords={'y': ['new1', 'new2']})
>>> compound_sum(x, c, 'x', 'c')
<xarray.DataArray (y: 2)>
array([40, 80])
Coordinates:
  * y        (y) <U4 'new1' 'new2'
```

xarray_extras.cumulatives.**compound_prod**(*x*, *c*, *xdim*, *cdim*)

Compound product among arbitrary points of x along dim See *compound_sum()*.

xarray_extras.cumulatives.**compound_mean**(*x*, *c*, *xdim*, *cdim*)

Compound mean among arbitrary points of x along dim See *compound_sum()*.

3.5 interpolate

xarray spline interpolation functions

xarray_extras.interpolate.**splrep**(*a*, *dim*, *k=3*)

Calculate the univariate B-spline for an N-dimensional array

Parameters

- **a** (*xarray.DataArray*) – any `DataArray`
- **dim** – dimension of a to be interpolated. `a.coords[dim]` must be strictly monotonic ascending. All int, float (not complex), or datetime dtypes are supported.
- **k** (*int*) – B-spline order:

k	interpolation kind
0	nearest neighbour
1	linear
2	quadratic
3	cubic

Returns `Dataset` with t, c, k (knots, coefficients, order) variables, the same shape and coords as the input, that can be passed to *splev()*.

Example:

```
>>> x = np.arange(0, 120, 20)
>>> x = xarray.DataArray(x, dims=['x'], coords={'x': x})
>>> s = xarray.DataArray(np.linspace(1, 20, 5), dims=['s'])
>>> y = np.exp(-x / s)
>>> x_new = np.arange(0, 120, 1)
>>> tck = splrep(y, 'x')
>>> y_new = splev(x_new, tck)
```

Features

- Interpolate a ND array on any arbitrary dimension
- dask supported on both on the interpolated array and x_new
- Supports ND x_new arrays
- The CPU-heavy interpolator generation (*splrep()*) is executed only once and then can be applied to multiple x_new (*splev()*)
- memory-efficient
- Can be pickled and used on dask distributed

Limitations

- Chunks are not supported along dim on the interpolated dimension.

`xarray_extras.interpolate.`**splev**(*x_new*, *tck*, *extrapolate=True*)

Evaluate the B-spline generated with *splrep()*.

Parameters

- **x_new** – Any `DataArray` with any number of dims, not necessarily the original interpolation dim. Alternatively, it can be any 1-dimensional array-like; it will be automatically converted to a `DataArray` on the interpolation dim.
- **tck** (*xarray.Dataset*) – As returned by *splrep()*. It can have been:
  - transposed (not recommended, as performance will drop if c is not C-contiguous)
  - sliced, reordered, or (re)chunked, on any dim except the interpolation dim
  - computed from dask to numpy backend
  - round-tripped to disk

- **extrapolate** –

  True Extrapolate the first and last polynomial pieces of b-spline functions active on the base interval

  False Return NaNs outside of the base interval

  'periodic' Periodic extrapolation is used

  'clip' Return y[0] and y[-1] outside of the base interval

Returns `DataArray` with all dims of the interpolated array, minus the interpolation dim, plus all dims of x_new

See *splrep()* for usage example.

3.6 numba_extras

Extensions to numba

xarray_extras.numba_extras.**guvectorize**(*signature*, *layout*, ***kwds*)

Convenience wrapper around `numba.guvectorize()`. Generate signature for all possible data types and set a few healthy defaults.

Parameters

- **signature** (*str*) – numba signature, containing {T}
- **layout** (*str*) – as in `numba.guvectorize()`
- **kwds** – passed verbatim to `numba.guvectorize()`. This function changes the default for cache from False to True.

example:

```
guvectorize("{T}[:], {T}[:]", "(i)->(i)")
```

Is the same as:

```
numba.guvectorize([
    "float32[:], float32[:]",
    "float64[:], float64[:]",
    ...
], "(i)->(i)", cache=True)
```

Note: Discussing upstream fix; see https://github.com/numba/numba/issues/2936.

3.7 recursive_diff

Recursively compare Python objects.

See also its most commonly used wrapper: *recursive_eq()*

xarray_extras.recursive_diff.**recursive_diff**(*lhs*, *rhs*, ***, *rel_tol=1e-09*, *abs_tol=0.0*, *brief_dims=()*)

Compare two objects and yield all differences. The two objects must any of:

- basic types (str, int, float, bool)
- basic collections (list, tuple, dict, set, frozenset)
- numpy.ndarray
- pandas.Series
- pandas.DataFrame
- pandas.Index
- xarray.DataArray
- xarray.Dataset
- any recursive combination of the above
- any other object (compared with ==)

Special treatment is reserved to different types:

- floats and ints are compared with tolerance, using math.isclose()
- NaN equals to NaN
- bools are only equal to other bools
- numpy arrays are compared elementwise and with tolerance, also testing the dtype
- pandas and xarray objects are compared elementwise, with tolerance, and without order, and do not support duplicate indexes
- xarray dimensions and variables are compared without order
- collections (list, tuple, dict, set, frozenset) are recursively descended into
- generic/unknown objects are compared with ==

Custom classes can be registered to benefit from the above behaviour; see documentation in *cast()*.

Parameters

- **lhs** – left-hand-side data structure
- **rhs** – right-hand-side data structure
- **rel_tol** (*float*) – relative tolerance when comparing numbers. Applies to floats, integers, and all numpy-based data.
- **abs_tol** (*float*) – absolute tolerance when comparing numbers. Applies to floats, integers, and all numpy-based data.
- **brief_dims** – One of:
  - sequence of strings representing xarray dimensions. If one or more differences are found along one of these dimensions, only one message will be reported, stating the differences count.
  - "all", to produce one line only for every xarray variable that differs

  Omit to output a line for every single different cell.

Yields strings containing difference messages, prepended by the path to the point that differs.

xarray_extras.recursive_diff.**cast**(*obj*, *brief_dims*)

Helper function of *recursive_diff()*.

Cast objects into simpler object types:

- Cast tuple to list
- Cast frozenset to set
- Cast all numpy-based objects to `xarray.DataArray`, as it is the most generic format that can describe all use cases:
  - `numpy.ndarray`
  - `pandas.Series`
  - `pandas.DataFrame`
  - `pandas.Index`, except `pandas.RangeIndex`, which is instead returned unaltered
  - `xarray.Dataset`

The data will be potentially wrapped by a dict to hold the various attributes and marked so that it doesn't trigger an infinite recursion.

- Do nothing for any other object types.

Parameters

- **obj** – complex object that must be simplified
- **brief_dims** (*tuple*) – sequence of xarray dimensions that must be compacted. See documentation on *recursive_diff()*.

Returns simpler object to compare

Custom objects

This is a single dispatch function which can be extended to compare custom objects. Take for example this custom class:

```
>>> class Rectangle:
...     def __init__(self, w, h):
...         self.w = w
...         self.h = h
...
...     def __eq__(self, other):
...         return self.w == other.w and self.h == other.h
...
...     def __repr__(self):
...         return 'Rectangle(%f, %f)' % (self.w, self.h)
```

The above can be processed by recursive_diff, because it supports the == operator against objects of the same type, and when converted to string it conveys meaningful information:

```
>>> list(recursive_diff(Rectangle(1, 2), Rectangle(3, 4)))
['Rectangle(1.000000, 2.000000) != Rectangle(2.000000, 3.000000)']
```

However, it doesn't support the more powerful features of recursive_diff, namely recursion and tolerance:

```
>>> list(recursive_diff(
...     Rectangle(1, 2), Rectangle(1.1, 2.2), abs_tol=.5))
['Rectangle(1.0000000, 2.0000000) != Rectangle(1.100000, 2.200000)']
```

This can be fixed by registering a custom cast function:

```
>>> @cast.register(Rectangle)
... def _(obj, brief_dims):
...     return {'w': obj.w, 'h': obj.h}
```

After doing so, w and h will be compared with tolerance and, if they are collections, will be recursively descended into:

```
>>> list(recursive_diff(
...     Rectangle(1, 2), Rectangle(1.1, 2.7), abs_tol=.5))
['[h]: 2.0 != 2.7 (abs: 7.0e-01, rel: 3.5e-01)']
```

3.8 sort

Sorting functions

xarray_extras.sort.**topk**(*a*, *k*, *dim*, *split_every=None*)

Extract the k largest elements from a on the given dimension, and return them sorted from largest to smallest. If k is negative, extract the -k smallest elements instead, and return them sorted from smallest to largest.

This assumes that k is small. All results will be returned in a single chunk along the given axis.

xarray_extras.sort.**argtopk**(*a*, *k*, *dim*, *split_every=None*)

Extract the indexes of the k largest elements from a on the given dimension, and return them sorted from largest to smallest. If k is negative, extract the -k smallest elements instead, and return them sorted from smallest to largest.

This assumes that k is small. All results will be returned in a single chunk along the given axis.

xarray_extras.sort.**take_along_dim**(*a*, *ind*, *dim*)

Use the output of *argtopk()* to pick points from a.

Parameters

- **a** – any xarray object
- **ind** – array of ints, as returned by *argtopk()*
- **dim** – dimension along which argtopk was executed

An example that uses all of the above functions is *source attribution*. Given a generic function $y = f(x_0, x_1, ..., x_i)$, which is embarassingly parallel along a given dimension, one wants to find:

- the top k elements of y along the dimension
- the elements of all x's that generated the top k elements of y

```
>>> from xarray import DataArray
>>> from xarray_extras.sort import *
>>> x = DataArray([[5, 3, 2, 8, 1],
>>>                [0, 7, 1, 3, 2]], dims=['x', 's'])
>>> y = x.sum('x')  # y = f(x), embarassingly parallel among dimension 's'
>>> y
<xarray.DataArray (s: 5)>
array([ 5, 10,  3, 11,  3])
Dimensions without coordinates: s
>>> top_y = topk(y, 3, 's')
>>> top_y
<xarray.DataArray (s: 3)>
```

(continues on next page)

(continued from previous page)

```
array([11, 10,  5])
Dimensions without coordinates: s
>>> top_x = take_along_dim(x, argtopk(y, 3, 's'), 's')
>>> top_x
<xarray.DataArray (x: 2, s: 3)>
array([[8, 3, 5],
       [3, 7, 0]])
Dimensions without coordinates: x, s
```

3.9 stack

Utilities for stacking/unstacking dimensions

xarray_extras.stack.**proper_unstack**(*array*, *dim*)

Work around an issue in xarray that causes the data to be sorted alphabetically by label on unstack():

https://github.com/pydata/xarray/issues/906

Also work around issue that causes string labels to be converted to objects:

https://github.com/pydata/xarray/issues/907

Parameters

- **array** – xarray.DataArray or xarray.Dataset to unstack
- **dim** (*str*) – Name of existing dimension to unstack

Returns xarray.DataArray / xarray.Dataset with unstacked dimension

3.10 testing

Tools for unit testing

xarray_extras.testing.**recursive_eq**(*lhs*, *rhs*, *rel_tol=1e-09*, *abs_tol=0.0*)

Wrapper around `recursive_diff()`. Print out all differences and finally assert that there are none.

3.11 ncdiff

Compare either two NetCDF files or all NetCDF files in two directories.

3.12 Usage

3.12.1 Chunking and RAM design

This tool does not support chunked files, or loading only part of large datasets into memory at once. Instead, chunked datasets are loaded as individual files. One variable at a time is then loaded into memory completely, compared, and then discarded.

This has the big advantage of simplicity, but a few disadvantages:

- No option to compare datasets with mismatched prefixes (e.g. `foo.*.nc` vs. `bar.*.nc`).
- No option to compare chunked datasets that differ only in chunking
- Slower, as there is no option to skip loading over and over again variables that don't sit on the concat_dim. See also xarray#2039.
- Huge RAM usage in case of monolithic variables

3.12.2 Further limitations

- Won't compare NetCDF settings, e.g. store version, compression, chunking, etc.
- Doesn't support indices with duplicate elements

CHAPTER 4

Credits

- *recursive_diff*, `recursive_eq()`, `proper_unstack()` and *ncdiff* were originally developed by Legal & General and released to the open source community in 2018.
- All boilerplate is from python_project_template, which in turn is from xarray.

CHAPTER 5

License

xarray-extras is available under the open source Apache License.

Python Module Index

x

xarray_extras.csv, 9
xarray_extras.cumulatives, 9
xarray_extras.interpolate, 10
xarray_extras.numba_extras, 12
xarray_extras.recursive_diff, 12
xarray_extras.sort, 15
xarray_extras.stack, 16
xarray_extras.testing, 16

Index

A

argtopk() (*in module xarray_extras.sort*), 15

C

cast() (*in module xarray_extras.recursive_diff*), 13
compound_mean() (*in module xarray_extras.cumulatives*), 10
compound_prod() (*in module xarray_extras.cumulatives*), 10
compound_sum() (*in module xarray_extras.cumulatives*), 10
cummean() (*in module xarray_extras.cumulatives*), 9

G

guvectorize() (*in module xarray_extras.numba_extras*), 12

P

proper_unstack() (*in module xarray_extras.stack*), 16

R

recursive_diff() (*in module xarray_extras.recursive_diff*), 12
recursive_eq() (*in module xarray_extras.testing*), 16

S

splev() (*in module xarray_extras.interpolate*), 11
splrep() (*in module xarray_extras.interpolate*), 10

T

take_along_dim() (*in module xarray_extras.sort*), 15
to_csv() (*in module xarray_extras.csv*), 9
topk() (*in module xarray_extras.sort*), 15

X

xarray_extras.csv (*module*), 9
xarray_extras.cumulatives (*module*), 9
xarray_extras.interpolate (*module*), 10
xarray_extras.numba_extras (*module*), 12
xarray_extras.recursive_diff (*module*), 12
xarray_extras.sort (*module*), 15
xarray_extras.stack (*module*), 16
xarray_extras.testing (*module*), 16