
dimarray Documentation

Release 1.0+0.g43fe5b8.dirty

Mahe Perrette

Oct 09, 2018

1	Introduction	3
1.1	Numpy array with dimensions	3
1.2	License	3
1.3	Getting started	3
1.4	Useful links	4
1.5	Install	4
1.6	Contributions	5
1.7	Packages you might also be interested in	5
2	Tutorial	7
2.1	define a dimarray	7
2.2	data structure	8
2.3	numpy-like attributes	8
2.4	indexing	8
2.5	transformation	9
2.6	alignment in operations	9
2.7	dataset	10
2.8	netCDF reading and writing	10
2.9	metadata	11
2.10	join arrays	11
2.11	drop missing data	12
2.12	reshaping arrays	13
2.13	interfacing with pandas	13
2.14	plotting	13
3	Advanced topics	17
3.1	Create a dimarray	17
3.2	Advanced Indexing	18
3.3	Along-axis transformations	21
3.4	Metadata	22
3.5	NetCDF reading and writing	23
4	Cookbook	29
4.1	Create a generic time-mean function	29
5	Talks	31

6	Reference API	33
6.1	DimArray API	33
6.2	Dataset API	60
6.3	Axis and Axes API	61
6.4	functions reference API	61
6.5	Table numpy vs dimarray	70
7	Maintainance of the documentation	73
8	Indices and tables	75

Welcome to dimarray documentation.

dimarray provides a *numpy* array with labelled axes and dimensions, in the spirit of *pandas* but generalized to N dimensions. It supports metadata and netCDF4 I/O.

1.1 Numpy array with dimensions

dimarray is a package to handle numpy arrays with labelled dimensions and axes. Inspired from pandas, it includes advanced alignment and reshaping features and as well as missing-value (NaN) handling.

The main difference with pandas is that it is generalized to N dimensions, and behaves more closely to a numpy array. The axes do not have fixed names ('index', 'columns', etc...) but are given a meaningful name by the user (e.g. 'time', 'items', 'lon' ...). This is especially useful for high dimensional problems such as sensitivity analyses.

A natural I/O format for such an array is netCDF, common in geophysics, which relies on the netCDF4 package, and supports metadata.

1.2 License

dimarray is distributed under a 3-clause ("Simplified" or "New") BSD license. Parts of basemap which have BSD compatible licenses are included. See the LICENSE file, which is distributed with the dimarray package, for details.

1.3 Getting started

A “**DimArray**” can be defined just like a numpy array, with additional information about its dimensions, which can be provided via its *axes* and *dims* parameters:

```
>>> from dimarray import DimArray
>>> a = DimArray([[1.,2,3], [4,5,6]], axes=[['a', 'b'], [1950, 1960, 1970]], dims=[
↳ 'variable', 'time'])
>>> a
```

(continues on next page)

(continued from previous page)

```
dimarray: 6 non-null elements (0 null)
0 / variable (2): 'a' to 'b'
1 / time (3): 1950 to 1970
array([[1., 2., 3.],
       [4., 5., 6.]])
```

Indexing now works on axes

```
>>> a['b', 1970]
6.0
```

Or can just be done **a la numpy**, via integer index:

```
>>> a.ix[0, -1]
3.0
```

Basic numpy transformations are also in there:

```
>>> a.mean(axis='time')
dimarray: 2 non-null elements (0 null)
0 / variable (2): 'a' to 'b'
array([2., 5.]])
```

Can export to *pandas* for pretty printing:

```
>>> a.to_pandas()
time      1950   1960   1970
variable
a          1.0    2.0    3.0
b          4.0    5.0    6.0
```

1.4 Useful links

Documentation	http://dimarray.readthedocs.org
Code on github (bleeding edge)	https://github.com/perrette/dimarray
Code on pypi (releases)	https://pypi.python.org/pypi/dimarray
Mailing List	http://groups.google.com/group/dimarray
Issues Tracker	https://github.com/perrette/dimarray/issues

1.5 Install

Requirements:

- python >= 2.7, 3
- numpy (tested with 1.7, 1.8, 1.9, 1.10.1, 1.15)

Optional:

- netCDF4 (tested with 1.0.8, 1.2.1) (netCDF archiving) (see notes below)
- matplotlib 1.1 (plotting)

- pandas 0.11 (interface with pandas)

Download the latest version from github and extract from archive Then from the dimarray repository type (possibly preceded by sudo):

```
python setup.py install
```

Alternatively, you can use pip to download and install the version from pypi (could be slightly out-of-date):

```
pip install dimarray
```

1.5.1 Notes on installing netCDF4

- On Ubuntu, using apt-get is the easiest way (as indicated at <https://github.com/Unidata/netcdf4-python/blob/master/.travis.yml>):

```
sudo apt-get install libhdf5-serial-dev netcdf-bin libnetcdf-dev
```

- On windows binaries are available: <http://www.unidata.ucar.edu/software/netcdf/docs/winbin.html>
- From source. Installing the netCDF4 python module from source can be cumbersome, because

it depends on netCDF4 and (especially) HDF5 C libraries that need to be compiled with specific flags (<http://unidata.github.io/netcdf4-python>). Detailed information on Ubuntu: <https://code.google.com/p/netcdf4-python/wiki/UbuntuInstall>

1.6 Contributions

All suggestions for improvement or direct contributions are very welcome. You can ask a question or start a discussion on the mailing list or open an *issue* on github for precise requests. See [links](#).

1.7 Packages you might also be interested in

`dimarray` is built on top of `numpy`, as an alternative to `larry` and `pandas`

`dimarray`'s default indexing method is on labels, which makes it very useful as data structure to store high dimensional problems with few labels, such as sensitivity analyses (or e.g. climate scenarios. . .).

If your focus is on large geoscientific data however, `xarray` is a more appropriate package, with useful methods to load large datasets, and a datamodel closely aligned on the netCDF. Moreover, standard, numpy-like integer indexing is more appropriate for geographic maps.

`pandas` is an excellent package for tabular data analysis, supporting many I/O formats and axis alignment (or “reindexing”) in binary operations. It is mostly limited to 2 dimensions (DataFrame), or up to 4 dimensions (Panel, Panel4D).

See also:

`DimArray.to_pandas()` and `DimArray.from_pandas()`.

[Download notebook](#)

- *define a dimarray*
- *data structure*
- *numpy-like attributes*
- *indexing*
- *transformation*
- *alignment in operations*
- *dataset*
- *netCDF reading and writing*
- *metadata*
- *join arrays*
- *drop missing data*
- *reshaping arrays*
- *interfacing with pandas*
- *plotting*

2.1 define a dimarray

A “**DimArray**” can be defined just like a numpy array, with additional information about its axes, which can be given via *axes* and *dims* parameters.

```
>>> from dimarray import DimArray, Dataset
>>> a = DimArray([[1.,2,3], [4,5,6]], axes=[['a', 'b'], [1950, 1960, 1970]], dims=[
↳ 'variable', 'time'])
>>> a
dimarray: 6 non-null elements (0 null)
0 / variable (2): 'a' to 'b'
1 / time (3): 1950 to 1970
array([[1., 2., 3.],
       [4., 5., 6.]])
```

2.2 data structure

Array data are stored in a *values* attribute:

```
>>> a.values
array([[1., 2., 3.],
       [4., 5., 6.]])
```

while its axes are stored in *axes*

```
>>> a.axes
0 / variable (2): 'a' to 'b'
1 / time (3): 1950 to 1970
```

As a convenience, axis labels can be accessed directly by name, as an alias for *a.axes['time'].values*:

```
>>> a.time
array([1950, 1960, 1970])
```

For more information refer to section on `page_data_structure_dimarray` (as well as `dimarray.Axis` and `dimarray.Axes`)

2.3 numpy-like attributes

Numpy-like attributes *dtype*, *shape*, *size* or *ndim* are defined, and are now augmented with *dims* and *labels*

```
>>> a.shape
(2, 3)
```

```
>>> a.dims      # grab axis names (the dimensions)
('variable', 'time')
```

```
>>> a.labels    # grab axis values
(array(['a', 'b'], dtype=object), array([1950, 1960, 1970]))
```

2.4 indexing

Indexing works on labels just as expected, including *slice* and boolean array.

```
>>> a['b', 1970]
6.0
```

but integer-index is always possible via *ix* toogle between *labels*- and *position*-based indexing:

```
>>> a.ix[0, -1]
3.0
```

See also:

Advanced Indexing

2.5 transformation

Standard numpy transformations are defined, and now accept axis name:

```
>>> a.mean(axis='time')
dimarray: 2 non-null elements (0 null)
0 / variable (2): 'a' to 'b'
array([2., 5.])
```

and can ignore **missing values (nans)** if asked to:

```
>>> import numpy as np
>>> a['a', 1950] = np.nan
>>> a.mean(axis='time', skipna=True)
dimarray: 2 non-null elements (0 null)
0 / variable (2): 'a' to 'b'
array([2.5, 5. ])
```

See also:

Along-axis transformations

2.6 alignment in operations

During an operation, arrays are **automatically re-indexed** to span the same axis domain, with nan filling if needed. This is quite useful when working with partly-overlapping time series or with incomplete sets of items.

```
>>> yearly_data = DimArray([0, 1, 2], axes=[[1950, 1960, 1970]], dims=['year'])
>>> incomplete_yearly_data = DimArray([10, 100], axes=[[1950, 1960]], dims=['year'])
↪ # last year 1970 is missing
>>> yearly_data + incomplete_yearly_data
dimarray: 2 non-null elements (1 null)
0 / year (3): 1950 to 1970
array([ 10., 101.,  nan])
```

See also:

reindex_axis, *reindex_like* and *align_axes*

A check is also performed on the dimensions, to ensure consistency of the data. If dimensions do not match this is not interpreted as an error but rather as a combination of dimensions. For example, you may want to combine some fixed spatial pattern (such as an EOF) with a time-varying time series (the principal component). Or you may want to combine results from a sensitivity analysis where several parameters have been varied (one dimension per parameter).

Here a minimal example where the above-defined annual variable is combined with seasonally-varying data (camping summer and winter prices).

Arrays are said to be **broadcast**:

```
>>> seasonal_data = DimArray([10, 100], axes=[['winter', 'summer']], dims=['season'])
>>> combined_data = yearly_data * seasonal_data
>>> combined_data
dimarray: 6 non-null elements (0 null)
0 / year (3): 1950 to 1970
1 / season (2): 'winter' to 'summer'
array([[ 0,  0],
       [ 10, 100],
       [ 20, 200]])
```

See also:

broadcast_arrays and *reshape*

2.7 dataset

Changed in version 0.1.9.

As a commodity, the **‘Dataset’** class is an ordered dictionary of DimArray instances which all share a common set of axes.

```
>>> dataset = Dataset({'combined_data':combined_data, 'yearly_data':yearly_data,
↳ 'seasonal_data':seasonal_data})
>>> dataset
Dataset of 3 variables
0 / season (2): 'winter' to 'summer'
1 / year (3): 1950 to 1970
seasonal_data: ('season',)
combined_data: ('year', 'season')
yearly_data: ('year',)
```

At initialization, the arrays are aligned on-the-fly. Later on, it is up to the user to reindex the arrays to match the Dataset axes.

Note: since Dataset elements share the same axes, any axis modification will also impact all contained DimArray instances. If this behaviour is not desired, a copy should be made.

2.8 netCDF reading and writing

A natural I/O format for such an array is netCDF, common in geophysics, which rely on the netCDF4 package. If netCDF4 is installed (much recommended), a dataset can easily read and write to the netCDF format:

```
>>> dataset.write_nc('/tmp/test.nc', mode='w')
```

```
>>> import dimarray as da
>>> da.read_nc('/tmp/test.nc', 'combined_data')
dimarray: 6 non-null elements (0 null)
```

(continues on next page)

(continued from previous page)

```
0 / year (3): 1950 to 1970
1 / season (2): u'winter' to u'summer'
array([[ 0,  0],
       [ 10, 100],
       [ 20, 200]])
```

See also:[NetCDF reading and writing](#)

2.9 metadata

It is possible to define and access metadata via the standard `.` syntax to access an object attribute:

```
>>> a = DimArray([1, 2])
```

```
>>> a.name = 'myarray'
>>> a.units = 'meters'
```

Any non-private attribute is automatically added to `a.attrs` ordered dictionary:

```
>>> a.attrs
OrderedDict([('name', 'myarray'), ('units', 'meters')])
```

Metadata can also be defined for `dimarray.Dataset` and `dimarray.Axis` instances, and will be written to / read from netCDF files.

Note: Metadata that start with an underscore `_` or use any protected class attribute as name (e.g. *values*, *axes*, *dims* and so on) must be set directly in `attrs`.

See also:[Metadata](#) for more information.

2.10 join arrays

DimArrays can be joined along an existing dimension, we say *concatenate* (`dimarray.concatenate()`):

```
>>> a = DimArray([11, 12, 13], axes=[[1950, 1951, 1952]], dims=['time'])
>>> b = DimArray([14, 15, 16], axes=[[1953, 1954, 1955]], dims=['time'])
>>> da.concatenate((a, b), axis='time')
dimarray: 6 non-null elements (0 null)
0 / time (6): 1950 to 1955
array([11, 12, 13, 14, 15, 16])
```

or they can be stacked along each other, thereby creating a new dimension (`dimarray.stack()`)

```
>>> a = DimArray([11, 12, 13], axes=[[1950, 1951, 1952]], dims=['time'])
>>> b = DimArray([21, 22, 23], axes=[[1950, 1951, 1952]], dims=['time'])
>>> da.stack((a, b), axis='items', keys=['a', 'b'])
dimarray: 6 non-null elements (0 null)
```

(continues on next page)

(continued from previous page)

```
0 / items (2): 'a' to 'b'
1 / time (3): 1950 to 1952
array([[11, 12, 13],
       [21, 22, 23]])
```

In the above note that new axis values were provided via the parameter *keys=*. If the common “time” dimension was not fully overlapping, array can be aligned prior to stacking via the *align=True* parameter.

```
>>> a = DimArray([11, 12, 13], axes=[[1950, 1951, 1952]], dims=['time'])
>>> b = DimArray([21, 23], axes=[[1950, 1952]], dims=['time'])
>>> c = da.stack((a, b), axis='items', keys=['a', 'b'], align=True)
>>> c
dimarray: 5 non-null elements (1 null)
0 / items (2): 'a' to 'b'
1 / time (3): 1950 to 1952
array([[11., 12., 13.],
       [21., nan, 23.]])
```

See also:

Join

2.11 drop missing data

Say you have data with NaNs:

```
>>> a = DimArray([[11, np.nan, np.nan], [21, np.nan, 23]], axes=[['a', 'b'], [1950, 1951, ↵
↵1952]], dims=['items', 'time'])
>>> a
dimarray: 3 non-null elements (3 null)
0 / items (2): 'a' to 'b'
1 / time (3): 1950 to 1952
array([[11., nan, nan],
       [21., nan, 23.]])
```

You can drop every column that contains a NaN

```
>>> a.dropna(axis=1) # drop along columns
dimarray: 2 non-null elements (0 null)
0 / items (2): 'a' to 'b'
1 / time (1): 1950 to 1950
array([[11.],
       [21.]])
```

or actually control decide to retain only these columns with a minimum number of valid data, here one:

```
>>> a.dropna(axis=1, minvalid=1) # drop every column with less than one valid data
dimarray: 3 non-null elements (1 null)
0 / items (2): 'a' to 'b'
1 / time (2): 1950 to 1952
array([[11., nan],
       [21., 23.]])
```

See also:

Missing values

2.12 reshaping arrays

Additional novelty includes methods to reshaping an array in easy ways, very useful for high-dimensional data analysis.

```
>>> large_array = DimArray(np.arange(2*2*5*2).reshape(2,2,5,2), dims=('A', 'B', 'C', 'D'
↪'))
>>> small_array = large_array.reshape('A,D', 'B,C')
>>> small_array
dimarray: 40 non-null elements (0 null)
0 / A,D (4): (0, 0) to (1, 1)
1 / B,C (10): (0, 0) to (1, 4)
array([[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18],
       [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19],
       [20, 22, 24, 26, 28, 30, 32, 34, 36, 38],
       [21, 23, 25, 27, 29, 31, 33, 35, 37, 39]])
```

See also:

[Modify shape](#) and [page_reshape](#)

2.13 interfacing with pandas

For things that pandas does better, such as pretty printing, I/O to many formats, and 2-D data analysis, just use the `dimarray.DimArray.to_pandas()` method. In the ipython notebook it also has a nice html rendering.

```
>>> small_array.to_pandas()
B      0      1
C      0      1      2      3      4      0      1      2      3      4
A D
0 0      0      2      4      6      8      10      12      14      16      18
   1      1      3      5      7      9      11      13      15      17      19
1 0      20      22      24      26      28      30      32      34      36      38
   1      21      23      25      27      29      31      33      35      37      39
```

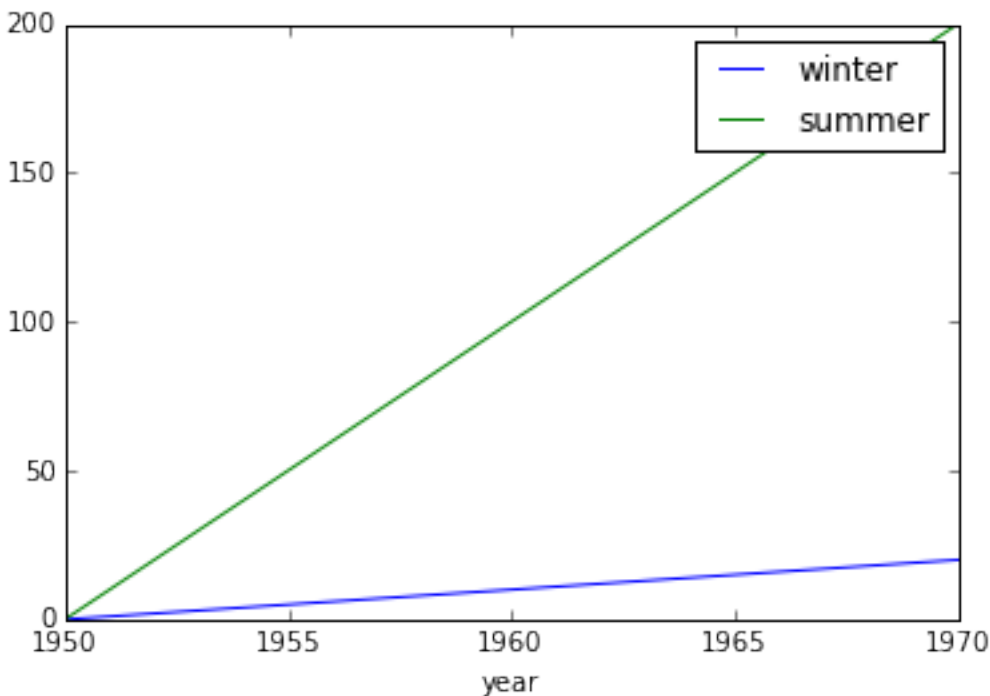
And `dimarray.DimArray.from_pandas()` works to convert pandas objects to `DimArray` (also supports *MultIndex*):

```
>>> import pandas as pd
>>> s = pd.DataFrame([[1,2],[3,4]], index=['a','b'], columns=[1950, 1960])
>>> da.from_pandas(s)
dimarray: 4 non-null elements (0 null)
0 / x0 (2): 'a' to 'b'
1 / x1 (2): 1950 to 1960
array([[1, 2],
       [3, 4]])
```

2.14 plotting

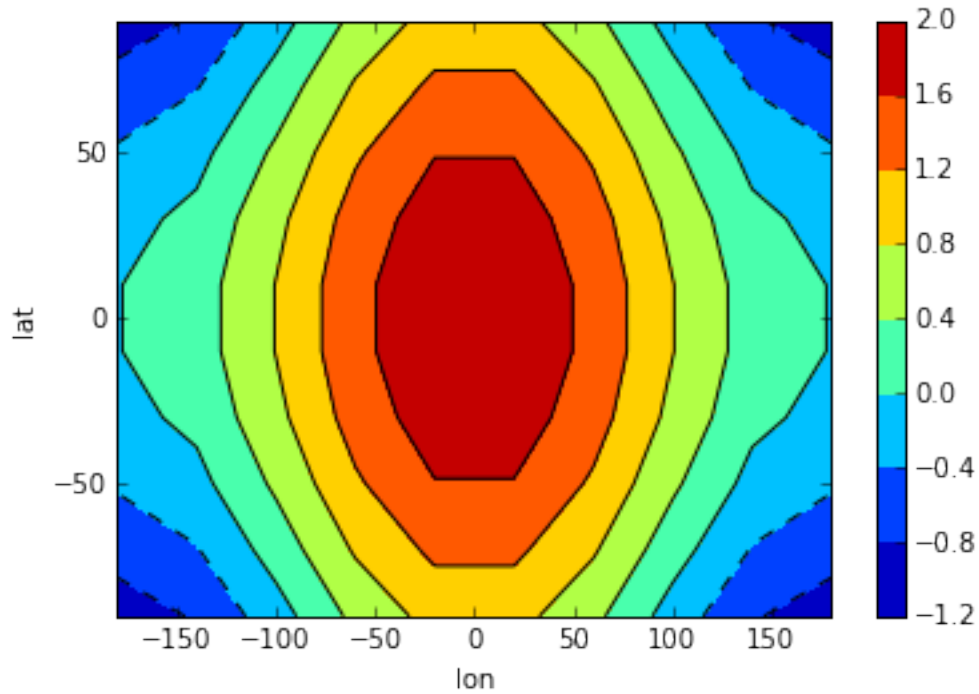
dimarray comes with basic plotting facility. For 1-D and 2-D data, it simplifies interfaces pandas' plot command (therefore pandas needs to be installed to use it). From the example above:

```
>>> %pylab # doctest: +SKIP
>>> %matplotlib inline # doctest: +SKIP
>>> a = dataset['combined_data']
>>> a.plot() # doctest: +SKIP
Using matplotlib backend: TkAgg
Populating the interactive namespace from numpy and matplotlib
[<matplotlib.lines.Line2D at 0x7f729cffdd50>,
 <matplotlib.lines.Line2D at 0x7f729cffded0>]
```

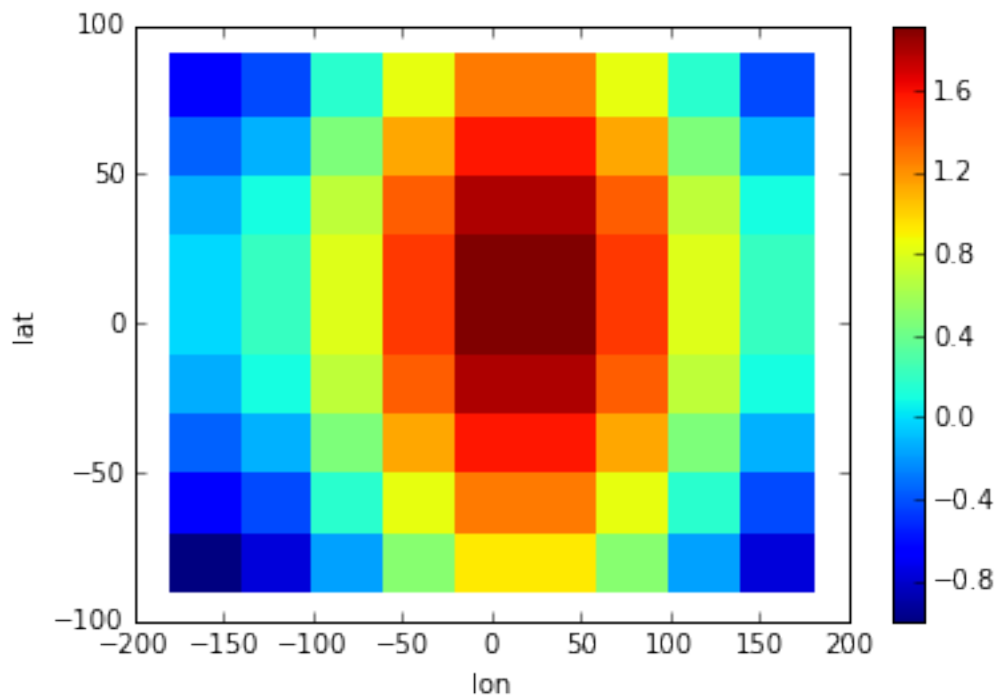


In addition, it can also display 2-D data via its methods *contour*, *contourf* and *pcolor* mapped from matplotlib.

```
>>> # create some data
>>> lon = np.linspace(-180, 180, 10)
>>> lat = np.linspace(-90, 90, 10)
>>> LON, LAT = np.meshgrid(lon, lat)
>>> DATA = np.cos(np.radians(LON)) + np.cos(np.radians(LAT))
>>> # define dimarray
>>> a = DimArray(DATA, axes=[lat, lon], dims=['lat', 'lon'])
>>> # plot the data
>>> c = a.contourf()
>>> colorbar(c) # explicit colorbar creation # doctest: +SKIP
>>> a.contour(colors='k') # doctest: +SKIP
/home/perrette/glacierenv/local/lib/python2.7/site-packages/matplotlib/collections.
↳py:650: FutureWarning: elementwise comparison failed; returning scalar instead, but
↳in the future will perform elementwise comparison
    if self._edgecolors_original != str('face'):
<matplotlib.contour.QuadContourSet instance at 0x7f729ce46b00>/home/perrette/
↳glacierenv/local/lib/python2.7/site-packages/matplotlib/collections.py:590:
↳FutureWarning: elementwise comparison failed; returning scalar instead, but in the
↳future will perform elementwise comparison
    if self._edgecolors == str('face'):
```



```
>>> # plot the data
>>> a.pcolor(colorbar=True) # colorbar as keyword argument # doctest: +SKIP
<matplotlib.collections.QuadMesh at 0x7f729cd3aa10>
```



For more information, you can use inline help (`help()` or `?`) or refer to [Advanced topics](#) and [Reference API](#)

Under construction...

3.1 Create a dimarray

[Download notebook](#)

There are various ways of defining a DimArray instance.

3.1.1 Standard definition

Provide a list of axis values (*axes*= parameter) and a list of axis names (*dims*=) parameter.

```
>>> from dimarray import DimArray
>>> a = DimArray([[1.,2,3], [4,5,6]], axes=[['a', 'b'], [1950, 1960, 1970]], dims=[
↳ 'variable', 'time'])
>>> a
dimarray: 6 non-null elements (0 null)
0 / variable (2): 'a' to 'b'
1 / time (3): 1950 to 1970
array([[1., 2., 3.],
       [4., 5., 6.]])
```

3.1.2 List of tuples

DimArray axes can also be initialized via a list of tuples (axis name, axis values):

```
>>> a = DimArray([[1.,2,3], [4,5,6]], axes=[('variable', ['a', 'b']), ('time', [1950,
↳ 1960, 1970])])
>>> a
dimarray: 6 non-null elements (0 null)
```

(continues on next page)

(continued from previous page)

```
0 / variable (2): 'a' to 'b'
1 / time (3): 1950 to 1970
array([[1., 2., 3.],
       [4., 5., 6.]])
```

3.1.3 Recursive definition : dict of dict

New in version 0.1.8.

It is possible to define a dimarray as a dictionary of dictionary. The only additional parameter needed is a list of dimension names, that should correspond to the dictionary's depth.

```
>>> dict_ = {'a': {1:11,
...               2:22,
...               3:33},
...         'b': {1:111,
...               2:222,
...               3:333} }
>>>
>>> DimArray(dict_, dims=['dim1', 'dim2'])
dimarray: 6 non-null elements (0 null)
0 / dim1 (2): 'a' to 'b'
1 / dim2 (3): 1 to 3
array([[ 11,  22,  33],
       [111, 222, 333]])
```

3.2 Advanced Indexing

[Download notebook](#)

Let's first define an array to test indexing

```
>>> from dimarray import DimArray
```

```
>>> v = DimArray([[1,2],[3,4],[5,6],[7,8]], axes=[["a","b","c","d"], [10.,20.]],
->dims=['x0','x1'], dtype=float)
>>> v
dimarray: 8 non-null elements (0 null)
0 / x0 (4): 'a' to 'd'
1 / x1 (2): 10.0 to 20.0
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])
```

3.2.1 Basics: integer, array, slice

There are various ways of indexing a DimArray, and all follow numpy's rules, except that in the default behaviour indices refer to axis values and not to position on the axis, in contrast to numpy.

```
>>> v['a',20] # extract a single item
2.0
```

The *ix* attribute is the pendant for position (integer) indexing (and exclusively so !). It is therefore similar to indexing on the *values* attribute, except that it returns a new DimArray, where `v.values[...]` would return a numpy ndarray.

```
>>> v.ix[0,:]
dimarray: 2 non-null elements (0 null)
0 / x1 (2): 10.0 to 20.0
array([1., 2.])
```

Note that the last element of slices is INCLUDED, contrary to numpy's position indexing. Step argument is always interpreted as an integer.

```
>>> v['a':'c',10] # 'c' is INCLUDED
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 'a' to 'c'
array([1., 3., 5.])
```

```
>>> v[['a','c'],10] # it is possible to provide a list
dimarray: 2 non-null elements (0 null)
0 / x0 (2): 'a' to 'c'
array([1., 5.])
```

```
>>> v[v.x0 != 'b',10] # boolean indexing is also fine
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 'a' to 'd'
array([1., 5., 7.])
```

If several array-like indices are provided, “orthogonal” indexing is performed, along each dimension independently:

```
>>> v[['a','c'],[10,20]] # it is possible to provide a list
dimarray: 4 non-null elements (0 null)
0 / x0 (2): 'a' to 'c'
1 / x1 (2): 10.0 to 20.0
array([[1., 2.],
       [5., 6.]])
```

See below for the cases where you do need numpy-like index broadcasting, using the *take* method.

3.2.2 Modify array values

All the above can be used to change array values, consistently with what you would expect.

```
>>> v['a':'c',10] = 11
>>> v.ix[2, -1] = 22 # same as v.values[2, -1] = 44
>>> v[v == 2] = 33
>>> v[v.x0 == 'b', v.x1 == 20] = 44
>>> v
dimarray: 8 non-null elements (0 null)
0 / x0 (4): 'a' to 'd'
1 / x1 (2): 10.0 to 20.0
array([[11., 33.],
       [11., 44.]])
```

(continues on next page)

(continued from previous page)

```
[11., 22.],
 [ 7.,  8.]])
```

3.2.3 take and put methods

These two methods `dimarray.DimArray.put()` and `dimarray.DimArray.take()` are the machinery to accessing and modifying items in the examples above. They may be useful to use directly for generic programming. They are similar to numpy methods of the same name, but also work in multiple dimensions. In particular, they both take dictionary, tuples and boolean arrays as *indices* argument.

```
>>> v = DimArray([[1,2],[3,4],[5,6],[7,8]], labels=[["a","b","c","d"], [10.,20.]],
↳dims=['x0','x1'], dtype=float)
```

```
>>> import numpy as np
>>> v[:,10] # doctest: +SKIP
>>> v.take(10, axis=1) # doctest: +SKIP
>>> v.take(10, axis='x1') # doctest: +SKIP
>>> v.take({'x1':10}) # dict # doctest: +SKIP
>>> v.take((slice(None),10)) # tuple # doctest: +SKIP
dimarray: 4 non-null elements (0 null)
0 / x0 (4): 'a' to 'd'
array([1., 3., 5., 7.])
```

The two latter forms, *tuple* or *dict*, allow performing multi-indexing. Array broadcasting is controlled by “broadcast” parameter.

```
>>> v.take({'x0':['a','b'], 'x1':[10, 20]}, broadcast=True)
dimarray: 2 non-null elements (0 null)
0 / x0,x1 (2): ('a', '10.0') to ('b', '20.0')
array([1., 4.])
```

```
>>> v.take({'x0':['a','b'], 'x1':[10, 20]}, broadcast=False) # same as v.box[['a','b'
↳'],[10, 20]]
dimarray: 4 non-null elements (0 null)
0 / x0 (2): 'a' to 'b'
1 / x1 (2): 10.0 to 20.0
array([[1., 2.],
       [3., 4.]])
```

The ‘indexing’ parameter can be set to *position* (same as *ix*) instead of *values*

```
>>> v.take(0, axis=1, indexing='position')
dimarray: 4 non-null elements (0 null)
0 / x0 (4): 'a' to 'd'
array([1., 3., 5., 7.])
```

Note the *put* command modifies values in-place by default, unless *inplace=False*.

```
>>> v.put(indices=10, values=-99, axis='x1', inplace=False)
dimarray: 8 non-null elements (0 null)
0 / x0 (4): 'a' to 'd'
1 / x1 (2): 10.0 to 20.0
array([[ -99.,   2.],
       [ -99.,   4.]])
```

(continues on next page)

(continued from previous page)

```
[-99.,  6.],
[-99.,  8.]])
```

3.3 Along-axis transformations

[Download notebook](#)

3.3.1 Basics

Most numpy transformations are built in. Let's create some data to try it out:

```
>>> from dimarray import DimArray
>>> a = DimArray([[1,2,3],[4,5,6]], axes=[['a','b'], [2000,2001,2002]], dims=['time',
↪ 'items'])
>>> a
dimarray: 6 non-null elements (0 null)
0 / time (2): 'a' to 'b'
1 / items (3): 2000 to 2002
array([[1, 2, 3],
       [4, 5, 6]])
```

The classical numpy syntax names are used (*sum*, *mean*, *max*...) are used. For transformation that reduce an axis, the default behaviour is to flatten the array prior to the transformation, consistently with numpy:

```
>>> a.mean() # sum over all axes
3.5
```

But the *axis=* parameter can also be passed explicitly to reduce only a specific axis:

```
>>> a.mean(axis=0) # sum over first axis
dimarray: 3 non-null elements (0 null)
0 / items (3): 2000 to 2002
array([2.5, 3.5, 4.5])
```

but it is now also possible to indicate axis name:

```
>>> a.mean(axis='time') # named axis
dimarray: 3 non-null elements (0 null)
0 / items (3): 2000 to 2002
array([2.5, 3.5, 4.5])
```

In addition, one can now provide a tuple to the *axis=* parameter, to reduce several axes at once

```
>>> a.mean(axis=('time','items')) # named axis
3.5
```

Of course, the above example makes more sense when they are more than two axes. To perform an operation on the flatten array, the convention is to provide the *None* value for *axis=*, which is the default behaviour in all reduction operators.

Note: transformations that accumulate along an axis (*cumsum*, *cumprod*) default on the last axis (*axis=-1*) instead of flattening the array. This is also true of the *diff* operator.

While most methods directly call numpy's in most cases, some subtle differences may exist that have to do with the need to define an axis values consistent with the operation. For example the `diff` method proposes several values for a `scheme` parameter ("centered", "backward", "forward"). Of interest also, the `argmin` and `argmax` methods return the value of the axis at the extrema instead of the integer position:

```
>>> a.argmax()
('a', 2000)
```

... which is consistent with dimarray indexing:

```
>>> a[a.argmax()]
1
```

3.3.2 Missing values

`dimarray` treats `NaN` as missing values, which can be skipped in transformations by passing `skipna=True`. In the example below we use a float-typed array because there is no `NaN` type in integer arrays.

```
>>> import numpy as np
>>> a = DimArray([[1,2,3],[4,5,6]], dtype=float)
>>> a[1,2] = np.nan
>>> a
dimarray: 5 non-null elements (1 null)
0 / x0 (2): 0 to 1
1 / x1 (3): 0 to 2
array([[ 1.,  2.,  3.],
       [ 4.,  5., nan]])
```

```
>>> a.sum(axis=0) # here the nans are not skipped
dimarray: 2 non-null elements (1 null)
0 / x1 (3): 0 to 2
array([ 5.,  7., nan])
```

```
>>> a.sum(axis=0, skipna=True)
dimarray: 3 non-null elements (0 null)
0 / x1 (3): 0 to 2
array([5., 7., 3.])
```

3.4 Metadata

Download notebook

`DimArray`, `Dataset` and `Axis` all support metadata. The straightforward way to define them is via the standard . syntax to access an object attribute:

```
>>> from dimarray import DimArray
>>> a = DimArray([1,2,3])
```

```
>>> a.name = 'distance'
>>> a.units = 'meters'
```

Although they are nothing more than usual python attributes, the `_metadata()` method gives an overview of all metadata:

```
>>> a.attrs
OrderedDict([('name', 'distance'), ('units', 'meters')])
```

Metadata are conserved by slicing and along-axis transformation, but are lost with more ambiguous operations.

```
>>> a[:].attrs
OrderedDict([('name', 'distance'), ('units', 'meters')])
```

```
>>> (a**2).attrs
OrderedDict()
```

Warning: The *attrs* attribute has been added in version 0.2, thereby deprecating the former *_metadata*.

A *summary()* method is also defined that provide an overview of both the data and its metadata.

```
>>> a.axes[0].units = 'axis units'
>>> a.summary()
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 0 to 2
    units: 'axis units'
attributes:
    name: 'distance'
    units: 'meters'
array([1, 2, 3])
```

Note: Metadata that start with an underscore *_* or use any protected class attribute as name (e.g. *values*, *axes*, *dims* and so on) can be set and accessed using by manipulating *attrs*.

```
>>> a.attrs['dims'] = 'this is a bad name'
```

```
>>> a.attrs
OrderedDict([('name', 'distance'), ('units', 'meters'), ('dims', 'this is a bad name
↪')])
```

```
>>> a.dims
('x0',)
```

It is easy to clear metadata:

```
>>> a.attrs = {} # clean all metadata
```

3.5 NetCDF reading and writing

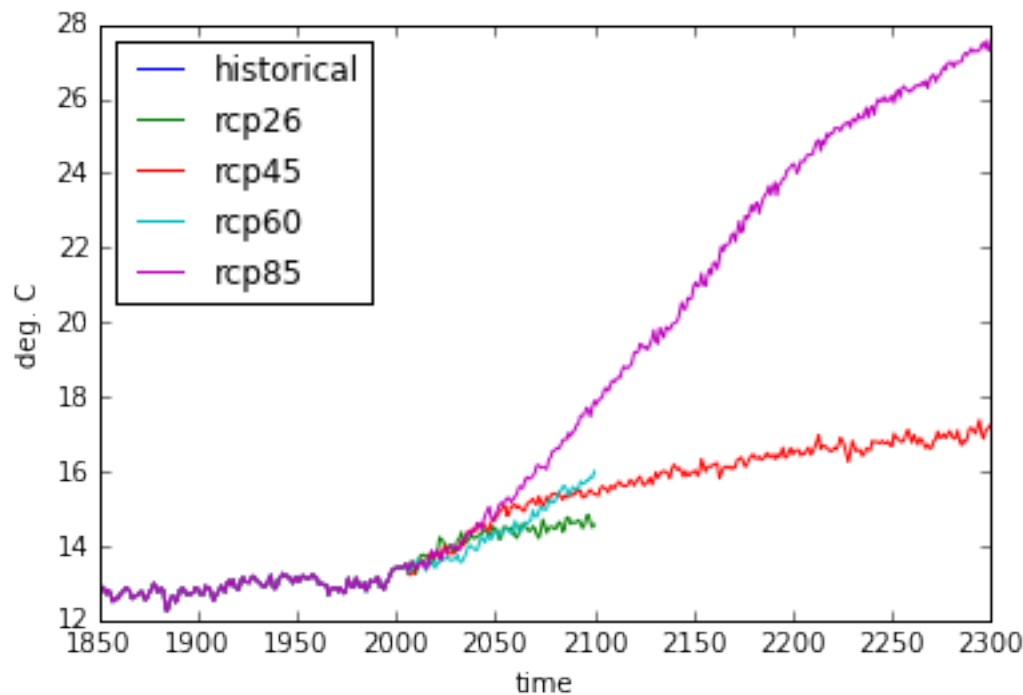
[Download notebook](#)

3.5.1 Read from one netCDF file

```
>>> from dimarray import read_nc, get_datadir
>>> import os
>>> ncfile = os.path.join(get_datadir(), 'cmip5.CSIRO-Mk3-6-0.nc') # get one netCDF_
↳file
>>> data = read_nc(ncfile) # load full file
>>> data
Dataset of 2 variables
0 / time (451): 1850 to 2300
1 / scenario (5): u'historical' to u'rcp85'
ts1: (u'time', u'scenario')
temp: (u'time', u'scenario')
```

Then access the variable of choice

```
>>> %pylab # doctest: +SKIP
>>> %matplotlib inline # doctest: +SKIP
>>> _ = data['temp'].plot()
>>> _ = plt.legend(loc='upper left') # doctest: +SKIP
Using matplotlib backend: TkAgg
Populating the interactive namespace from numpy and matplotlib
```



Load only one variable

```
>>> data = read_nc(ncfile, 'temp') # only one variable
>>> data = read_nc(ncfile, 'temp', indices={"time": slice(2000, 2100), "scenario": "rcp45"}
↳)}) # load only a chunk of the data
>>> data = read_nc(ncfile, 'temp', indices={"time": 1950.3}, tol=0.5) # approximate_
↳matching, adjust tolerance
>>> data = read_nc(ncfile, 'temp', indices={"time": -1}, indexing='position') # _
↳integer position indexing
```

3.5.2 Read from multiple files

Read variable ‘temp’ across multiple files (representing various climate models). In this case the variable is a time series, whose length may vary across experiments (thus align=True is passed to reindex axes before stacking). Under the hood the function `py:func:dimarray.stack` is called:

```
>>> direc = get_datadir()
>>> temp = read_nc(direc+'/cmip5.*.nc', 'temp', align=True, axis='model')
```

A new ‘model’ axis is created labeled with file names. It is then possible to rename it more appropriately, e.g. keeping only the part directly relevant to identify the experiment:

```
>>> getmodel = lambda x: os.path.basename(x).split('.')[1] # extract model name from
↳ path
>>> temp.set_axis(getmodel, axis='model', inplace=True) # would return a copy if
↳ inplace is not specified
>>> temp
dimarray: 9114 non-null elements (6671 null)
0 / model (7): 'CSIRO-Mk3-6-0' to 'MPI-ESM-MR'
1 / time (451): 1850 to 2300
2 / scenario (5): u'historical' to u'rcp85'
array(...)
```

This works on datasets as well

```
>>> ds = read_nc(direc+'/cmip5.*.nc', align=True, axis='model')
>>> ds.set_axis(getmodel, axis='model', inplace=True)
>>> ds
Dataset of 2 variables
0 / model (7): 'CSIRO-Mk3-6-0' to 'MPI-ESM-MR'
1 / time (451): 1850 to 2300
2 / scenario (5): u'historical' to u'rcp85'
ts1: ('model', u'time', u'scenario')
temp: ('model', u'time', u'scenario')
```

3.5.3 Write to netCDF

Let’s define some dummy arrays representing temperature in northern and southern hemisphere for three years.

```
>>> from dimarray import DimArray
>>> temperature = DimArray([[1.,2,3], [4,5,6]], axes=[['north','south'], [1951, 1952,
↳ 1953]], dims=['lat', 'time'])
>>> global_mean = temperature.mean(axis='lat')
>>> climatology = temperature.mean(axis='time')
```

Let’s define a new dataset

```
>>> from dimarray import Dataset
>>> ds = Dataset({'temperature':temperature, 'global':global_mean})
>>> ds
Dataset of 2 variables
0 / time (3): 1951 to 1953
1 / lat (2): 'north' to 'south'
global: ('time',)
temperature: ('lat', 'time')
```

Saving the dataset to file is pretty simple:

```
>>> ds.write_nc('/tmp/test.nc', mode='w')
```

It is possible to append more variables

```
>>> climatology.write_nc('/tmp/test.nc', 'climatology', mode='a') # by default mode=
↪ 'w'
```

Just as a check, all three variables seem to be there:

```
>>> read_nc('/tmp/test.nc')
Dataset of 3 variables
0 / time (3): 1951 to 1953
1 / lat (2): u'north' to u'south'
global: (u'time',)
temperature: (u'lat', u'time')
climatology: (u'lat',)
```

Note that when appending a variable to a netCDF file or to a dataset, its axes must match, otherwise an error will be raised. In that case it may be necessary to reindex an axis (see [page_reindexing](#)). When initializing a dataset with bunch of dimarray however, reindexing is performed automatically.

3.5.4 New NetCDF4 storage

New in version 0.2.

Since version 0.2, the methods above are a wrapper around :class:dimarray.DatasetOnDisk class, which allows lower level access with a DimArray feeling.

```
>>> import dimarray as da
>>> import numpy as np
>>> dima = da.DimArray([[1,2,3],[4,5,6]], axes=[('time',[2000,2045.5]),('scenario',['a
↪ ','b','c'])])
>>> dima.units = 'myunits' # metadata
>>> dima.axes['time'].units = 'metadata-dim-in-memory'
>>>
>>> ds = da.open_nc('/tmp/test.nc', mode='w')
>>> ds['myvar'] = dima
>>> ds['myvar'].bla = 'bla'
>>> ds['myvar'].axes['time'].yo = 'metadata-dim-on-disk'
>>> ds.axes['scenario'].ya = 'metadata-var-on-disk'
>>> ds.yi = 'metadata-dataset-on-disk'
>>> ds.close()
```

Let's check the result:

```
>>> ds2 = da.open_nc("/tmp/test.nc", mode="a")
>>> ds2
DatasetOnDisk of 1 variable (NETCDF4)
0 / time (2): 2000.0 to 2045.5
1 / scenario (3): u'a' to u'c'
myvar: (u'time', u'scenario')
```

```
>>> ds2.summary()
DatasetOnDisk of 1 variable (NETCDF4)
<BLANKLINE>
//dimensions:
```

(continues on next page)

(continued from previous page)

```

0 / time (2): 2000.0 to 2045.5
  units: u'metadata-dim-in-memory'
  yo: u'metadata-dim-on-disk'
1 / scenario (3): u'a' to u'c'
  ya: u'metadata-var-on-disk'
<BLANKLINE>
//variables:
myvar: (u'time', u'scenario')
      units: u'myunits'
      bla: u'bla'
<BLANKLINE>
//global attributes:
  yi: u'metadata-dataset-on-disk'

```

```

>>> ds2['myvar']
DimArrayOnDisk: 'myvar' (6)
0 / time (2): 2000.0 to 2045.5
1 / scenario (3): u'a' to u'c'

```

```

>>> ds2['myvar'].values # doctest: +SKIP
<type 'netCDF4._netCDF4.Variable'>
int64 myvar(time, scenario)
      units: myunits
      bla: bla
unlimited dimensions:
current shape = (2, 3)
filling on, default _FillValue of -9223372036854775806 used

```

```

>>> ds2['myvar'][:]
dimarray: 6 non-null elements (0 null)
0 / time (2): 2000.0 to 2045.5
1 / scenario (3): u'a' to u'c'
array([[1, 2, 3],
       [4, 5, 6]])

```

```

>>> ds2['myvar'][2000, 'b'] = 77
>>> ds2['myvar'][:]
dimarray: 6 non-null elements (0 null)
0 / time (2): 2000.0 to 2045.5
1 / scenario (3): u'a' to u'c'
array([[1, 77, 3],
       [4, 5, 6]])

```

```

>>> ds2['myvar'].ix[0, -1] = -1
>>> ds2['myvar'][:]
dimarray: 6 non-null elements (0 null)
0 / time (2): 2000.0 to 2045.5
1 / scenario (3): u'a' to u'c'
array([[1, 77, -1],
       [4, 5, 6]])

```

```
>>> ds2.close()
```

Create a variable with unlimited dimension

```
>>> import dimarray as da
>>>
>>> ds = da.open_nc('/tmp/test.nc', 'w')
>>> ds.axes.append('time', None)
>>> ds.nc.dimensions['time'] # underlying netCDF4 object # doctest: +SKIP
<type 'netCDF4._netCDF4.Dimension'> (unlimited): name = 'time', size = 0
```

Fill-up the variable:

```
>>> ds['bla'] = da.DimArray([1,2,3,4,5], dims=['time'], axes=[list('abcde')])
>>> ds.nc.dimensions['time'] # underlying netCDF4 object # doctest: +SKIP
<type 'netCDF4._netCDF4.Dimension'> (unlimited): name = 'time', size = 5
```

Append some new slices:

```
>>> ds['bla'].ix[5] = da.DimArray([66], dims=['time'], axes=[list('f')])
>>> ds.nc.dimensions['time'] # underlying netCDF4 object # doctest: +SKIP
<type 'netCDF4._netCDF4.Dimension'> (unlimited): name = 'time', size = 6
```

```
>>> ds['bla'].read()
dimarray: 6 non-null elements (0 null)
0 / time (6): u'a' to u'f'
array([ 1,  2,  3,  4,  5, 66])
```

```
>>> ds.close()
```

4.1 Create a generic time-mean function

[Download notebook](#)

This function applies to any kind of input array, as long as the “time” dimension is present.

```
>>> def time_mean(a, t1=None, t2=None):
...     """ compute time mean between two instants
...
...     Parameters:
...     -----
...     a : DimArray
...         must include a "time" dimension
...     t1, t2 : same type as a.time (typically int or float)
...         start and end times
...
...     Returns:
...     -----
...     ma : DimArray
...         time-average between t1 and t2
...     """
...     assert 'time' in a.dims, 'dimarray must have the "time" dimension'
...     return a.swapaxes(0, 'time')[t1:t2].mean(axis='time')
```

```
>>> from dimarray import DimArray
>>> import numpy as np
```

```
>>> a = DimArray([1,2,3,4], axes=[[2000,2001,2002,2003]], dims=['time'])
>>> time_mean(a, 2001, 2003) # average over 2001, 2002, 2003
3.0
```

```
>>> a = DimArray([[1,2,3,4],[5,6,7,8]], axes=[['a','b'],[2000,2001,2002,2003]], dims=[
↳ 'items','time'])
>>> time_mean(a) # average over the full time axis
dimarray: 2 non-null elements (0 null)
0 / items (2): 'a' to 'b'
array([2.5, 6.5])
```

CHAPTER 5

Talks

Selected talks involving dimarray:

- Modelling strategy seminar at PIK (May 2014): [Brief introduction to python, numpy and dimarray](#)

Under construction...

Classical functions have been organized in categories. For a reference documentation please see inline help and next section.

```
>>> help(DimArray.diff) # doctest: +SKIP
```

or with *ipython*

```
>>> DimArray.diff? # doctest: +SKIP
```

6.1 DimArray API

DimArray methods are list below by topic, along with examples. Functions are provided in a separate page [functions reference API](#).

- *Create a DimArray*
- *Modify shape*
- *Reduce, accumulate*
- *Indexing*
- *Re-indexing*
- *Missing values*
- *To / From other objects*
- *I/O*
- *Plotting*

6.1.1 Create a DimArray

`DimArray.__init__`(*values=None, axes=None, dims=None, labels=None, copy=False, dtype=None, _indexing=None, _indexing_broadcast=None, **kwargs*)

Initialize a DimArray instance

Parameters

values [numpy-like array, or DimArray instance, or dict] If *values* is not provided, will initialize an empty array with dimensions inferred from *axes* (in that case *axes=* must be provided).

axes [list or tuple, optional] axis values as ndarrays, whose order matches axis names (the dimensions) provided via *dims=* parameter. Each axis can also be provided as a tuple (str, array-like) which contains both axis name and axis values, in which case *dims=* becomes superfluous. *axes=* can also be provided with a list of Axis objects. If *axes=* is omitted, a standard axis `np.arange(shape[i])` is created for each axis *i*.

dims [list or tuple, optional] dimensions (or axis names) This parameter can be omitted if dimensions are already provided by other means, such as passing a list of tuple to *axes=*. If axes are passed as keyword arguments (via ****kwargs**), *dims=* is used to determine the order of dimensions. If *dims* is not provided by any of the means mentioned above, default dimension names are given *x0, x1, ... 'xn'*, where *n* is the number of dimensions.

dtype [numpy data type, optional] passed to `np.array()`

copy [bool, optional] passed to `np.array()`

****kwargs** [keyword arguments] metadata

Notes

metadata passed this way cannot have name already taken by other parameters such as “values”, “axes”, “dims”, “dtype” or “copy”.

Examples

Basic:

```
>>> DimArray([[1,2,3],[4,5,6]]) # automatic labelling
dimarray: 6 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (3): 0 to 2
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> DimArray([[1,2,3],[4,5,6]], dims=['items','time']) # axis names only
dimarray: 6 non-null elements (0 null)
0 / items (2): 0 to 1
1 / time (3): 0 to 2
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> DimArray([[1,2,3],[4,5,6]], axes=[list("ab"), np.arange(1950,1953)]) # axis_
↪ values only
dimarray: 6 non-null elements (0 null)
0 / x0 (2): 'a' to 'b'
```

(continues on next page)

(continued from previous page)

```
1 / x1 (3): 1950 to 1952
array([[1, 2, 3],
       [4, 5, 6]])
```

More general case:

```
>>> a = DimArray([[1,2,3],[4,5,6]], axes=[list("ab"), np.arange(1950,1953)],
↳dims=['items','time'])
>>> b = DimArray([[1,2,3],[4,5,6]], axes=[('items',list("ab")), ('time',np.
↳arange(1950,1953))])
>>> c = DimArray([[1,2,3],[4,5,6]], {'items':list("ab"), 'time':np.arange(1950,
↳1953)}) # here dims can be omitted because shape = (2, 3)
>>> np.all(a == b) and np.all(a == c)
True
>>> a
dimarray: 6 non-null elements (0 null)
0 / items (2): 'a' to 'b'
1 / time (3): 1950 to 1952
array([[1, 2, 3],
       [4, 5, 6]])
```

Empty data

```
>>> a = DimArray(axes=[('items',list("ab")), ('time',np.arange(1950,1953))])
```

Metadata

```
>>> a = DimArray([[1,2,3],[4,5,6]], name='test', units='none')
```

6.1.2 Modify shape

`DimArray.transpose(*dims)`

Permute dimensions

Analogous to numpy, but also allows axis names

Parameters

***dims** [int or str] variable list of dimensions

Returns

transposed_array [DimArray]

See also:

reshape, *flatten*, *unflatten*, *newaxis*

Examples

```
>>> import dimarray as da
>>> a = da.DimArray(np.zeros((2,3)), ['x0','x1'])
>>> a
dimarray: 6 non-null elements (0 null)
```

(continues on next page)

(continued from previous page)

```
0 / x0 (2): 0 to 1
1 / x1 (3): 0 to 2
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> a.T
dimarray: 6 non-null elements (0 null)
0 / x1 (3): 0 to 2
1 / x0 (2): 0 to 1
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> (a.T == a.transpose(1,0)).all() and (a.T == a.transpose('x1','x0')).all()
True
```

`DimArray.swapaxes(axis1, axis2)`

Swap two axes

analogous to numpy's `swapaxes`, but can provide axes by name

Parameters

axis1, axis2 [int or str] axes to swap (transpose)

Returns

transposed_array [DimArray]

Examples

```
>>> from dimarray import DimArray
>>> a = DimArray(np.arange(2*3*4).reshape(2,3,4))
>>> a.dims
('x0', 'x1', 'x2')
>>> b = a.swapaxes('x2',0) # put 'x2' at the first position
>>> b.dims
('x2', 'x1', 'x0')
>>> b.shape
(4, 3, 2)
```

`DimArray.reshape(*newdims, **kwargs)`

Add/remove/flatten dimensions to conform array to new dimensions

Parameters

newdims [tuple or list or variable list of dimension names {str}] Any dimension now present in the array is added as singleton dimension Any dimension name containing a comma is interpreting as a flattening command. All dimensions to flatten have to exist already.

transpose [bool] if True, transpose dimensions to match new order (default True) otherwise, raise and Error if tranpose is needed (closer to original numpy's behaviour)

Returns

reshaped_array [DimArray] with `reshaped_array.dims == tuple(newdims)`

See also:

flatten, unflatten, transpose, newaxis

Examples

```
>>> from dimarray import DimArray
>>> a = DimArray([7,8])
>>> a
dimarray: 2 non-null elements (0 null)
0 / x0 (2): 0 to 1
array([7, 8])
```

```
>>> a.reshape(('x0', 'new'))
dimarray: 2 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / new (1): None to None
array([[7],
       [8]])
```

```
>>> b = DimArray(np.arange(2*2*2).reshape(2,2,2))
>>> b
dimarray: 8 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (2): 0 to 1
2 / x2 (2): 0 to 1
array([[[0, 1],
        [2, 3]],
       <BLANKLINE>
        [[4, 5],
        [6, 7]]])
```

```
>>> c = b.reshape('x0', 'x1,x2')
>>> c
dimarray: 8 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1,x2 (4): (0, 0) to (1, 1)
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
>>> c.reshape('x0,x1', 'x2')
dimarray: 8 non-null elements (0 null)
0 / x0,x1 (4): (0, 0) to (1, 1)
1 / x2 (2): 0 to 1
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

`DimArray.flatten(*dims, **kwargs)`

Flatten all or a subset of dimensions

Parameters

dims [list or tuple of axis names, optional] by default, all dimensions

reverse [bool, optional] if True, reverse behaviour: dims are interpreted as the dimensions to keep, and all the other dimensions are flattened default is False

insert [int, optional] position where to insert the flattened axis (by default, any flattened dimension is inserted at the position of the first axis involved in flattening)

Returns

flattened_array [DimArray] appropriately reshaped, with collapsed dimensions as first axis (tuples)

This is useful to do a regional mean with missing values

See also:

reshape, transpose

Notes

A tuple of axis names can be passed via the “axis” parameter of the transformation to trigger flattening prior to reducing an axis.

Examples

Flatten all dimensions

```
>>> from dimarray import DimArray
>>> a = DimArray([[1, 2, 3], [4, 5, 6]])
>>> a
dimarray: 6 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (3): 0 to 2
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> b = a.flatten()
>>> b
dimarray: 6 non-null elements (0 null)
0 / x0,x1 (6): (0, 0) to (1, 2)
array([1, 2, 3, 4, 5, 6])
```

```
>>> b.labels
(array([(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)], dtype=object),)
```

Flatten a subset of dimensions only

```
>>> from dimarray import DimArray
>>> np.random.seed(0)
>>> values = np.arange(2*3*4).reshape(2, 3, 4)
>>> v = DimArray(values, axes=[('time', [1950, 1955]), ('lat', np.linspace(-90, 90,
↪3)), ('lon', np.linspace(-180, 180, 4))])
>>> v
dimarray: 24 non-null elements (0 null)
0 / time (2): 1950 to 1955
1 / lat (3): -90.0 to 90.0
2 / lon (4): -180.0 to 180.0
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
<BLANKLINE>
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
```

```
>>> w = v.flatten(('lat', 'lon'), insert=1)
>>> w
dimarray: 24 non-null elements (0 null)
0 / time (2): 1950 to 1955
1 / lat,lon (12): (-90.0, -180.0) to (90.0, 180.0)
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]])
```

```
>>> np.all( w.unflatten() == v )
True
```

But be careful, the order matter !

```
>>> v.flatten(('lon', 'lat'), insert=1)
dimarray: 24 non-null elements (0 null)
0 / time (2): 1950 to 1955
1 / lon,lat (12): (-180.0, -90.0) to (180.0, 90.0)
array([[ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11],
       [12, 16, 20, 13, 17, 21, 14, 18, 22, 15, 19, 23]])
```

Useful to average over a group of dimensions:

```
>>> v.flatten(('lon', 'lat'), insert=0).mean(axis=0)
dimarray: 2 non-null elements (0 null)
0 / time (2): 1950 to 1955
array([ 5.5, 17.5])
```

is equivalent to:

```
>>> v.mean(axis=('lon', 'lat'))
dimarray: 2 non-null elements (0 null)
0 / time (2): 1950 to 1955
array([ 5.5, 17.5])
```

`DimArray.unflatten` (*axis=None*)
undo flatten (inflate array)

Parameters

axis [int or str or None, optional] axis to unflatten default to None to unflatten all

Returns

DimArray

`DimArray.squeeze` (*axis=None*)
Squeeze singleton axes

Analogous to numpy, but also allows axis name

Parameters

axis [int or str or None] axis to squeeze default is None, to remove all singleton axes

Returns

squeezed_array [DimArray]

Examples

```
>>> import dimarray as da
>>> a = da.DimArray([[[[1,2,3]]]])
>>> a
dimarray: 3 non-null elements (0 null)
0 / x0 (1): 0 to 0
1 / x1 (1): 0 to 0
2 / x2 (3): 0 to 2
array([[[[1, 2, 3]]]])
>>> a.squeeze()
dimarray: 3 non-null elements (0 null)
0 / x2 (3): 0 to 2
array([1, 2, 3])
>>> a.squeeze(axis='x1')
dimarray: 3 non-null elements (0 null)
0 / x0 (1): 0 to 0
1 / x2 (3): 0 to 2
array([[1, 2, 3]])
```

`DimArray.repeat` (*values*, *axis=None*)
expand the array along an existing axis

Parameters

values [int or ndarray or Axis instance] int: size of new axis ndarray: values of new axis

axis [int or str] refer to the dimension along which to repeat

****kwaxes** [key-word arguments] alternatively, axes may be passed as keyword arguments

Returns

DimArray

See also:

`newaxis`

Examples

```
>>> import dimarray as da
>>> a = da.DimArray(np.arange(3), labels = [[1950., 1951., 1952.]], dims=('time',
↪))
>>> a2d = a.newaxis('lon', pos=1) # lon is now singleton dimension
```

```
>>> a2d.repeat(2, axis="lon")
dimarray: 6 non-null elements (0 null)
0 / time (3): 1950.0 to 1952.0
1 / lon (2): 0 to 1
array([[0, 0],
       [1, 1],
       [2, 2]])
```

```
>>> a2d.repeat([30., 50.], axis="lon")
dimarray: 6 non-null elements (0 null)
0 / time (3): 1950.0 to 1952.0
1 / lon (2): 30.0 to 50.0
array([[0, 0],
       [1, 1],
       [2, 2]])
```

`DimArray.broadcast` (*other*)

repeat array to match target dimensions

Parameters

other [DimArray or Axes objects or ordered Dictionary of axis values]

Returns

DimArray

Examples

Create some dummy data: # ...create some dummy data:

```
>>> import dimarray as da
>>> lon = np.linspace(10, 30, 2)
>>> lat = np.linspace(10, 50, 3)
>>> time = np.arange(1950, 1955)
>>> ts = da.DimArray(np.arange(5), axes=[time], dims=['time'])
>>> cube = da.DimArray(np.zeros((3, 2, 5)), axes=[('lat', lat), ('lon', lon), ('time',
↪time)]) # lat x lon x time
>>> cube.axes
0 / lat (3): 10.0 to 50.0
1 / lon (2): 10.0 to 30.0
2 / time (5): 1950 to 1954
```

...broadcast timeseries to 3D data

```
>>> ts3D = ts.broadcast(cube) # lat x lon x time
>>> ts3D
dimarray: 30 non-null elements (0 null)
0 / lat (3): 10.0 to 50.0
1 / lon (2): 10.0 to 30.0
2 / time (5): 1950 to 1954
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]],
<BLANKLINE>
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]],
<BLANKLINE>
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
```

6.1.3 Reduce, accumulate

`DimArray.max` (*skipna=False*, *args=()*, ***kwargs*)

Analogous to numpy's max

`max(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If True, treat NaN as missing values (either using `MaskedArray` or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

`DimArray`, or numpy array or scalar (e.g. in some cases if ‘axis’ is *None*)

See help on `numpy.max` or `numpy.ma.max` for other parameters and more information.

See also:

`apply_along_axis` is called by this method

`to_MaskedArray` is used if `skipna` is True

`DimArray.min(skipna=False, args=(), **kwargs)`

Analogous to numpy’s `min`

`min(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If True, treat NaN as missing values (either using `MaskedArray` or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

`DimArray`, or numpy array or scalar (e.g. in some cases if ‘axis’ is *None*)

See help on `numpy.min` or `numpy.ma.min` for other parameters

and more information.

See also:

`apply_along_axis` is called by this method

`to_MaskedArray` is used if `skipna` is `True`

`DimArray.ptp` (*skipna=False*, *args=()*, ***kwargs*)

Analogous to numpy's `ptp`

`ptp(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If **True**, treat NaN as missing values (either using **MaskedArray** or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is *None*)

See help on `numpy.ptp` or `numpy.ma.ptp` for other parameters

and more information.

See also:

`apply_along_axis` is called by this method

`to_MaskedArray` is used if `skipna` is `True`

`DimArray.median` (*skipna=False*, *args=()*, ***kwargs*)

Analogous to numpy's `median`

`median(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If True, treat NaN as missing values (either using MaskedArray or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is None)

See help on numpy.median or numpy.ma.median for other parameters and more information.

See also:

apply_along_axis is called by this method

to_MaskedArray is used if skipna is True

DimArray.**all** (*skipna=False*, *args=()*, ***kwargs*)

Analogous to numpy’s all

all(..., axis=None, skipna=False, ...)

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If True, treat NaN as missing values (either using MaskedArray or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is None)

See help on numpy.all or numpy.ma.all for other parameters and more information.

See also:

apply_along_axis is called by this method

to_MaskedArray is used if skipna is True

`DimArray.any(skipna=False, args=(), **kwargs)`

Analogous to numpy's `any`

`any(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If **True**, treat NaN as missing values (either using **MaskedArray** or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is *None*)

See help on `numpy.any` or `numpy.ma.any` for other parameters

and more information.

See also:

apply_along_axis is called by this method

to_MaskedArray is used if `skipna` is **True**

`DimArray.prod(skipna=False, args=(), **kwargs)`

Analogous to numpy's `prod`

`prod(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If **True**, treat NaN as missing values (either using **MaskedArray** or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is None)

See help on `numpy.prod` or `numpy.ma.prod` for other parameters
and more information.

See also:

`apply_along_axis` is called by this method

`to_MaskedArray` is used if `skipna` is True

`DimArray.sum(skipna=False, args=(), **kwargs)`

Analogous to numpy’s `sum`

`sum(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If True, treat NaN as missing values (either using `MaskedArray` or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is None)

See help on `numpy.sum` or `numpy.ma.sum` for other parameters
and more information.

See also:

`apply_along_axis` is called by this method

`to_MaskedArray` is used if `skipna` is True

`DimArray.mean(skipna=False, args=(), **kwargs)`

Analogous to numpy’s `mean`

`mean(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If **True**, treat NaN as missing values (either using **MaskedArray** or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is *None*)

See help on **numpy.mean** or **numpy.ma.mean** for other parameters and more information.

See also:

apply_along_axis is called by this method

to_MaskedArray is used if skipna is **True**

DimArray.std(*skipna=False*, *args=()*, ***kwargs*)

Analogous to numpy’s std

std(..., axis=None, skipna=False, ...)

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If **True**, treat NaN as missing values (either using **MaskedArray** or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is *None*)

See help on **numpy.std** or **numpy.ma.std** for other parameters and more information.

See also:

apply_along_axis is called by this method

`to_MaskedArray` is used if `skipna` is `True`

`DimArray.var` (*skipna=False, args=(), **kwargs*)

Analogous to numpy's `var`

`var(..., axis=None, skipna=False, ...)`

Accepts the same parameters as the equivalent numpy function, with modified behaviour of the *axis* parameter and an additional *skipna* parameter to handle NaNs (by default considered missing values)

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If **True**, treat NaN as missing values (either using **MaskedArray** or, when available, specific numpy function)

“...” stands for any other parameters required by the function, and depends on the particular function being called

Returns

DimArray, or numpy array or scalar (e.g. in some cases if ‘axis’ is *None*)

See help on `numpy.var` or `numpy.ma.var` for other parameters and more information.

See also:

`apply_along_axis` is called by this method

`to_MaskedArray` is used if `skipna` is `True`

`DimArray.argmax` (*axis=None, skipna=False*)

similar to numpy's `argmax`, but return axis values instead of integer position

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If **True**, treat NaN as missing values (either using **MaskedArray** or, when available, specific numpy function)

`DimArray.argmin` (*axis=None, skipna=False*)

similar to numpy's `argmin`, but return axis values instead of integer position

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *None*.

skipna [bool]

If True, treat NaN as missing values (either using MaskedArray or, when available, specific numpy function)

`DimArray.cumsum(axis=-1, skipna=False)`

`DimArray.cumprod(axis=-1, skipna=False)`

`DimArray.diff(axis=-1, scheme='backward', keepaxis=False, n=1)`

Analogous to numpy's `diff`

Calculate the *n*-th order discrete difference along given axis.

The first order difference is given by $\text{out}[n] = a[n+1] - a[n]$ along the given axis, higher order differences are calculated by using *diff* recursively.

Parameters

axis [int or str or tuple] axis along which to apply the transform. Can be given as axis position (*int*), as axis name (*str*), as a *list* or *tuple* of axes (positions or names) to collapse into one axis before applying transform. If *axis* is *None*, just apply the transform on the flattened array consistently with numpy (in this case will return a scalar). Default is *-1*.

scheme [str, optional] determines the values of the resulting axis - “forward”: $\text{diff}[i] = x[i+1] - x[i]$ - “backward”: $\text{diff}[i] = x[i] - x[i-1]$ - “centered”: $\text{diff}[i] = x[i+1/2] - x[i-1/2]$ Default is “backward”

keepaxis [bool, optional] if True, keep the initial axis by padding with NaNs Only compatible with “forward” or “backward” differences Default is False

n [int, optional] The number of times values are differenced. Default is one

Returns

diff [DimArray] The *n* order differences. The shape of the output is the same as *a* except along *axis* where the dimension is smaller by *n*.

Examples

Create some example data

```
>>> import dimarray as da
>>> v = da.DimArray([1,2,3,4], ('time', np.arange(1950,1954)), dtype=float)
>>> s = v.cumsum()
>>> s
dimarray: 4 non-null elements (0 null)
0 / time (4): 1950 to 1953
array([ 1.,  3.,  6., 10.]
```

diff reduces axis size by one, by default

```
>>> s.diff()
dimarray: 3 non-null elements (0 null)
0 / time (3): 1951 to 1953
array([2., 3., 4.])
```

The `keepaxis=` parameter fills array with `nan` where necessary to keep the axis unchanged. Default is backward differencing: $\text{diff}[i] = v[i] - v[i-1]$.

```
>>> s.diff(keepaxis=True)
dimarray: 3 non-null elements (1 null)
0 / time (4): 1950 to 1953
array([nan, 2., 3., 4.])
```

But other schemes are available to control how the new axis is defined: *backward* (default), *forward* and even *centered*

```
>>> s.diff(keepaxis=True, scheme="forward") # diff[i] = v[i+1] - v[i]
dimarray: 3 non-null elements (1 null)
0 / time (4): 1950 to 1953
array([ 2., 3., 4., nan])
```

The `keepaxis=True` option is invalid with the *centered* scheme, since every axis value is modified by definition:

```
>>> s.diff(axis='time', scheme='centered')
dimarray: 3 non-null elements (0 null)
0 / time (3): 1950.5 to 1952.5
array([2., 3., 4.])
```

6.1.4 Indexing

`DimArray.__getitem__` (*indices=None, axis=0, indexing=None, tol=None, broadcast=None, keep-dims=False, broadcast_arrays=None*)

`DimArray.ix()`

`DimArray.box()`
property to allow indexing without array broadcasting (matlab-like)

`DimArray.take()`
Retrieve values from a DimArray

Parameters

indices [int or list or slice (single-dimensional indices)] or a tuple of those (multi-dimensional) or *dict* of { axis name : axis values }

axis [None or int or str, optional] if specified and indices is a slice, scalar or an array, assumes indexing is along this axis.

indexing [{‘label’, ‘position’}, optional] Indexing mode. - “label”: indexing on axis labels (default) - “position”: use numpy-like position index Default value can be changed in `dimarray.rcParams[‘indexing.by’]`

tol [None or float or tuple or dict, optional] tolerance when looking for numerical values, e.g. to use nearest neighbor search, default *None*.

keepdims [bool, optional] keep singleton dimensions (default False)

broadcast [bool, optional] if True, use numpy-like *fancy* indexing and broadcast any indexing array to a common shape, useful for example to sample points along a path. Default to False.

Returns

indexed_array [DimArray instance or scalar]

See also:

`DimArray.put`, `DimArrayOnDisk.read`, `DimArray.take_axis`

Examples

```
>>> from dimarray import DimArray
>>> v = DimArray([[1,2,3],[4,5,6]], axes=["a","b"], [10.,20.,30.], dims=['d0',
↪ 'd1'], dtype=float)
>>> v
dimarray: 6 non-null elements (0 null)
0 / d0 (2): 'a' to 'b'
1 / d1 (3): 10.0 to 30.0
array([[1., 2., 3.],
       [4., 5., 6.]])
```

Indexing via axis values (default)

```
>>> a = v[:,10] # python slicing method
>>> a
dimarray: 2 non-null elements (0 null)
0 / d0 (2): 'a' to 'b'
array([1., 4.])
>>> b = v.take(10, axis=1) # take, by axis position
>>> c = v.take(10, axis='d1') # take, by axis name
>>> d = v.take({'d1':10}) # take, by dict {axis name : axis values}
>>> (a==b).all() and (a==c).all() and (a==d).all()
True
```

Indexing via integer index (indexing="position" or ix property)

```
>>> np.all(v.ix[:,0] == v[:,10])
True
>>> np.all(v.take(0, axis="d1", indexing="position") == v.take(10, axis="d1"))
True
```

Multi-dimensional indexing

```
>>> v["a", 10] # also work with string axis
1.0
>>> v.take(('a',10)) # multi-dimensional, tuple
1.0
>>> v.take({'d0':'a', 'd1':10}) # dict-like arguments
1.0
```

Take a list of indices

```
>>> a = v[:, [10, 20]] # also work with a list of index
>>> a
dimarray: 4 non-null elements (0 null)
0 / d0 (2): 'a' to 'b'
1 / d1 (2): 10.0 to 20.0
array([[1., 2.],
       [4., 5.]])
>>> b = v.take([10, 20], axis='d1')
>>> np.all(a == b)
True
```

Take a slice:

```
>>> c = v[:, 10:20] # axis values: slice includes last element
>>> c
dimarray: 4 non-null elements (0 null)
0 / d0 (2): 'a' to 'b'
1 / d1 (2): 10.0 to 20.0
array([[1., 2.],
       [4., 5.]])
>>> d = v.take(slice(10, 20), axis='d1') # `take` accepts `slice` objects
>>> np.all(c == d)
True
>>> v.ix[:, 0:1] # integer position: does not include last element
dimarray: 2 non-null elements (0 null)
0 / d0 (2): 'a' to 'b'
1 / d1 (1): 10.0 to 10.0
array([[1.],
       [4.]])
```

Keep dimensions

```
>>> a = v[["a"]]
>>> b = v.take("a", keepdims=True)
>>> np.all(a == b)
True
```

tolerance parameter to achieve “nearest neighbour” search

```
>>> v.take(12, axis="d1", tol=5)
dimarray: 2 non-null elements (0 null)
0 / d0 (2): 'a' to 'b'
array([1., 4.] )
```

Matlab like multi-indexing

```
>>> v = DimArray(np.arange(2*3*4).reshape(2, 3, 4))
>>> v[[0, 1], :, [0, 0, 0]].shape
(2, 3, 3)
>>> v[[0, 1], :, [0, 0]].shape # here broadcast = False
(2, 3, 2)
>>> v.take(([0, 1], slice(None), [0, 0]), broadcast=True).shape # that is traditional_
↪numpy, with broadcasting on same shape
(2, 3)
>>> v.values[[0, 1], :, [0, 0]].shape # a proof of it
(2, 3)
```

```
>>> a = DimArray(np.arange(2*3).reshape(2,3))
```

```
>>> a[a > 3] # FULL ARRAY: return a numpy array in n-d case (at least for now)
dimarray: 2 non-null elements (0 null)
0 / x0,x1 (2): (1, 1) to (1, 2)
array([4, 5])
```

```
>>> a[a.x0 > 0] # SINGLE AXIS: only first axis
dimarray: 3 non-null elements (0 null)
0 / x0 (1): 1 to 1
1 / x1 (3): 0 to 2
array([[3, 4, 5]])
```

```
>>> a[:, a.x1 > 0] # only second axis
dimarray: 4 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (2): 1 to 2
array([[1, 2],
       [4, 5]])
```

```
>>> a[a.x0 > 0, a.x1 > 0]
dimarray: 2 non-null elements (0 null)
0 / x0 (1): 1 to 1
1 / x1 (2): 1 to 2
array([[4, 5]])
```

Sample points along a path, a la numpy, with broadcast=True

```
>>> a.take([(0,0,1],[1,2,2]), broadcast=True)
dimarray: 3 non-null elements (0 null)
0 / x0,x1 (3): (0, 1) to (1, 2)
array([1, 2, 5])
```

Ellipsis (only one supported)

```
>>> a = DimArray(np.arange(2*3*4*5).reshape(2,3,4,5))
>>> a[0,...,0].shape
(3, 4)
>>> a[... ,0,0].shape
(2, 3)
```

`DimArray.put()`

Modify values of a DimArray

Parameters

indices [int or list or slice (single-dimensional indices)] or a tuple of those (multi-dimensional) or *dict* of { axis name : axis values }

axis [None or int or str, optional] if specified and indices is a slice, scalar or an array, assumes indexing is along this axis.

indexing [{‘label’, ‘position’}, optional] Indexing mode. - ‘label’: indexing on axis labels (default) - ‘position’: use numpy-like position index Default value can be changed in `dimarray.rcParams[‘indexing.by’]`

tol [None or float or tuple or dict, optional] tolerance when looking for numerical values, e.g. to use nearest neighbor search, default *None*.

broadcast [bool, optional] if True, use numpy-like *fancy* indexing and broadcast any indexing array to a common shape, useful for example to sample points along a path. Default to False.

Returns

None (inplace=True) or DimArray instance or scalar (inplace=False)

See also:

`DimArray.take`, `DimArrayOnDisk.write`

6.1.5 Re-indexing

`DimArray.reset_axis` (*axis=0, **kwargs*)

`DimArray.reindex_axis` (*values, axis=0, fill_value=nan, raise_error=False, method=None*)

reindex an array along an axis

Parameters

values [array-like or Axis] new axis values

axis [int or str, optional] axis number or name

fill_value: bool, optional Fill data to use for missing axis value, if *raise_error* is False.

raise_error [bool, optional] if True, raise error when an axis value is not present otherwise just replace with *fill_value*. Default is False

method [[None, 'left', 'right']] method to fill the gaps (default None) If 'left' or 'right', just pass along to `numpy.searchsorted`.

Returns

dimarray: DimArray instance

Examples

Basic reindexing: fill missing values with NaN

```
>>> import dimarray as da
>>> a = da.DimArray([1,2,3], axes=[('x0', [1,2,3])])
>>> b = da.DimArray([3,4], axes=[('x0', [1,3])])
>>> b.reindex_axis([1,2,3])
dimarray: 2 non-null elements (1 null)
0 / x0 (3): 1 to 3
array([ 3., nan,  4.] )
```

Or replace with anything else, like -9999

```
>>> b.reindex_axis([1,2,3], fill_value=-9999)
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 1 to 3
array([ 3, -9999,  4])
```

`DimArray.reindex_like` (*other*, ***kwargs*)

`reindex_like` : re-index like another dimarray / axes instance

Applies `reindex_axis` on each axis to match another `DimArray`

Parameters

other [DimArray or Axes instance]

****kwargs** :

Returns

DimArray

Notes

only reindex axes which are present in other

Examples

```
>>> import dimarray as da
>>> b = da.DimArray([3,4], ('x0', [1,3]))
>>> c = da.DimArray([[1,2,3], [1,2,3]], (('x1', ["a", "b"]), ('x0', [1, 2, 3])))
>>> b.reindex_like(c)
dimarray: 2 non-null elements (1 null)
0 / x0 (3): 1 to 3
array([ 3., nan,  4.])
```

`DimArray.sort_axis` (*axis=0*, *key=None*, *kind='quicksort'*)

sort an axis

Parameters

a [DimArray (this argument is pre-assigned when using as bound method)]

axis [int or str, optional] axis by position (int) or name (str) (default: 0)

key [callable or dict-like, optional] function that is called on each axis label and whose return value is used for sorting instead of axis label. Any other object with `__getitem__` attribute may also be used as key, such as a dictionary. If `None` (the default), axis label is used for sorting.

kind [str, optional] sort algorithm (see `numpy.sort` for more info)

Returns

sorted [new DimArray with sorted axis]

Examples

Basic

```
>>> from dimarray import DimArray
>>> a = DimArray([10,20,30], labels=[2, 0, 1])
>>> a
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 2 to 1
array([10, 20, 30])
```

```
>>> a.sort_axis()
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 0 to 2
array([20, 30, 10])
```

```
>>> a.sort_axis(key=lambda x: -x)
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 2 to 0
array([10, 30, 20])
```

Multi-dimensional

```
>>> a = DimArray([[10,20,30],[40,50,60]], labels=[[0, 1], ['a','c','b']])
>>> a.sort_axis(axis=1)
dimarray: 6 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (3): 'a' to 'c'
array([[10, 30, 20],
       [40, 60, 50]])
```

6.1.6 Missing values

`DimArray.dropna` (*axis=0*, *minvalid=None*, *na=nan*)

drop nans along an axis

Parameters

axis [axis position or name or list of names]

minvalid [int, optional] min number of valid point in each slice along axis values by default all the points

Returns

DimArray

Examples

1-Dimension

```
>>> from dimarray import DimArray
>>> a = DimArray([1.,2,3], ('time',[1950, 1955, 1960]))
>>> a.ix[1] = np.nan
>>> a
dimarray: 2 non-null elements (1 null)
0 / time (3): 1950 to 1960
array([ 1., nan,  3.])
>>> a.dropna()
```

(continues on next page)

(continued from previous page)

```
dimarray: 2 non-null elements (0 null)
0 / time (2): 1950 to 1960
array([1., 3.]
```

Multi-dimensional

```
>>> a = DimArray([[ np.nan, 2., 3.],[ np.nan, 5., np.nan]])
>>> a
dimarray: 3 non-null elements (3 null)
0 / x0 (2): 0 to 1
1 / x1 (3): 0 to 2
array([[nan, 2., 3.],
       [nan, 5., nan]])
>>> a.dropna(axis=1)
dimarray: 2 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (1): 1 to 1
array([[2.],
       [5.]])
>>> a.dropna(axis=1, minvalid=1) # minimum number of valid values, equivalent to_
↳ `how="all"` in pandas
dimarray: 3 non-null elements (1 null)
0 / x0 (2): 0 to 1
1 / x1 (2): 1 to 2
array([[ 2., 3.],
       [ 5., nan]])
```

`DimArray.fillna(value, inplace=False, na=nan)`
Fill NaN with a replacement value

Examples

```
>>> from dimarray import DimArray
>>> a = DimArray([1,2,np.nan])
>>> a.fillna(-99)
dimarray: 3 non-null elements (0 null)
0 / x0 (3): 0 to 2
array([ 1., 2., -99.]
```

`DimArray.setna(value, na=nan, inplace=False)`
set a value as missing

Parameters

- value** [the values to set to na]
- na** [the replacement value (default np.nan)]

Examples

```
>>> from dimarray import DimArray
>>> a = DimArray([1,2,-99])
>>> a.setna(-99)
dimarray: 2 non-null elements (1 null)
```

(continues on next page)

(continued from previous page)

```

0 / x0 (3): 0 to 2
array([ 1.,  2., nan])
>>> a.setna([-99, 2]) # sequence
dimarray: 1 non-null elements (2 null)
0 / x0 (3): 0 to 2
array([ 1., nan, nan])
>>> a.setna(a > 1) # boolean
dimarray: 2 non-null elements (1 null)
0 / x0 (3): 0 to 2
array([ 1., nan, -99.])
>>> a = DimArray([[1,2,-99]]) # multi-dim
>>> a.setna([-99, a>1]) # boolean
dimarray: 1 non-null elements (2 null)
0 / x0 (1): 0 to 0
1 / x1 (3): 0 to 2
array([[ 1., nan, nan]])

```

6.1.7 To / From other objects

classmethod `DimArray.from_pandas` (*data*, *dims=None*)

Initialize a DimArray from pandas

Parameters

data [pandas object (Series, DataFrame, Panel, Panel4D)]

dims, optional [dimension (axis) names, otherwise look at ax.name for ax in data.axes]

Returns

a [DimArray instance]

Examples

```

>>> import pandas as pd
>>> s = pd.Series([3,5,6], index=['a','b','c'])
>>> s.index.name = 'dim0'
>>> DimArray.from_pandas(s)
dimarray: 3 non-null elements (0 null)
0 / dim0 (3): 'a' to 'c'
array([3, 5, 6])

```

Also work with Multi-Index

```

>>> panel = pd.Panel(np.arange(2*3*4).reshape(2,3,4))
>>> b = panel.to_frame() # pandas' method to convert Panel to DataFrame via_
↳MultiIndex
>>> DimArray.from_pandas(b) # doctest: +SKIP
dimarray: 24 non-null elements (0 null)
0 / major,minor (12): (0, 0) to (2, 3)
1 / x1 (2): 0 to 1
...

```

`DimArray.to_pandas()`
return the equivalent pandas object

`DimArray.to_larry()`
return the equivalent pandas object

`DimArray.to_dataset(axis=0)`
split a DimArray into a Dataset object (collection of DimArrays)

6.1.8 I/O

`DimArray.write_nc(f, name=None, mode='w', clobber=None, format=None, *args, **kwargs)`
Write to netCDF

Parameters

f [file name]

name [variable name, optional] must be provided if no attribute “name” is defined

mode, clobber, format [see netCDF4.Dataset]

****kwargs** [passed to netCDF4.Dataset.createVariable (compression)]

See also:

`DatasetOnDisk`

6.1.9 Plotting

`DimArray.plot(*args, **kwargs)`
Plot 1-D or 2-D data.

Wraps matplotlib’s plot()

Parameters

***args, **kwargs** [passed to matplotlib.pyplot.plot]

legend [True (default) or False] Display legend for 2-D data.

ax [matplotlib.Axis, optional] Provide axis on which to show the plot.

Returns

lines [list of matplotlib’s Lines2D instances]

Examples

```
>>> from dimarray import DimArray
>>> data = DimArray(np.random.rand(4,3), axes=[np.arange(4), ['a','b','c']],
↳dims=['distance', 'label'])
>>> data.axes[0].units = 'meters'
>>> h = data.plot(linewidth=2)
>>> h = data.T.plot(linestyle='-.')
>>> h = data.plot(linestyle='-.', legend=False)
```

`DimArray.pcolor(*args, **kwargs)`

Plot a quadrilateral mesh.

Wraps matplotlib `pcolormesh()`. See `pcolormesh` documentation in matplotlib for accepted keyword arguments.

Examples

```
>>> from dimarray import DimArray
>>> x = DimArray(np.zeros([100,40]))
>>> x.pcolor() # doctest: +SKIP
>>> x.T.pcolor() # to flip horizontal/vertical axes # doctest: +SKIP
```

`DimArray.contourf(*args, **kwargs)`

Plot filled 2-D contours.

Wraps matplotlib `contourf()`. See `contourf` documentation in matplotlib for accepted keyword arguments.

Examples

```
>>> from dimarray import DimArray
>>> x = DimArray(np.zeros([100,40]))
>>> x[:50,:20] = 1.
>>> x.contourf() # doctest: +SKIP
>>> x.T.contourf() # to flip horizontal/vertical axes # doctest: +SKIP
```

`DimArray.contour(*args, **kwargs)`

Plot 2-D contours.

Wraps matplotlib `contour()`. See `contour` documentation in matplotlib for accepted keyword arguments.

Examples

```
>>> from dimarray import DimArray
>>> x = DimArray(np.zeros([100,40]))
>>> x[:50,:20] = 1.
>>> x.contour() # doctest: +SKIP
>>> x.T.contour() # to flip horizontal/vertical axes # doctest: +SKIP
```

6.2 Dataset API

Under construction...

6.3 Axis and Axes API

6.3.1 Axis

class dimarray.**Axis**

Under construction...

6.3.2 Axes

class dimarray.**Axes**

Under construction...

6.3.3 GroupedAxis reference API

class dimarray.**GroupedAxis**

Under construction...

6.4 functions reference API

dimarray functions are listed below by topic, along with examples. DimArray Methods are provided in a separate page *DimArray API*.

- *Join*
- *Align*
- *Interpolate*
- *Stats*
- *Read netCDF data*
- *dimarray options*

6.4.1 Join

dimarray.**stack** (*arrays*, *axis=None*, *keys=None*, *align=False*, ***kwargs*)

stack arrays along a new dimension (raise error if already existing)

Parameters

arrays [sequence or dict of arrays]

axis [str, optional] new dimension along which to stack the array

keys [array-like, optional] stack axis values, useful if array is a sequence, or a non-ordered dictionary

align [bool, optional] if True, align axes prior to stacking (Default to False)

****kwargs** [optional key-word arguments passed to align, if align is True]

Returns

DimArray [joint array]

See also:

`concatenate` join arrays along an existing dimension

`swapaxes` to modify the position of the newly inserted axis

Examples

```
>>> from dimarray import DimArray
>>> a = DimArray([1,2,3])
>>> b = DimArray([11,22,33])
>>> stack([a, b], axis='stackdim', keys=['a', 'b'])
dimarray: 6 non-null elements (0 null)
0 / stackdim (2): 'a' to 'b'
1 / x0 (3): 0 to 2
array([[ 1,  2,  3],
       [11, 22, 33]])
```

`dimarray.concatenate` (*arrays*, *axis=0*, *_no_check=False*, *align=False*, ***kwargs*)
concatenate several DimArrays

Parameters

arrays [list of DimArrays] arrays to concatenate

axis [int or str] axis along which to concatenate (must exist)

align [bool, optional] align secondary axes before joining on the primary axis *axis*. Default to False.

*****kwargs*** [optional key-word arguments passed to align, if align is True]

Returns

concatenated DimArray

See also:

`stack` join arrays along a new dimension

`align` align arrays

Examples

1-D

```
>>> from dimarray import DimArray
>>> a = DimArray([1,2,3], axes=[['a', 'b', 'c']])
>>> b = DimArray([4,5,6], axes=[['d', 'e', 'f']])
>>> concatenate((a, b))
dimarray: 6 non-null elements (0 null)
0 / x0 (6): 'a' to 'f'
array([1, 2, 3, 4, 5, 6])
```

2-D

```

>>> a = DimArray([[1,2,3],[11,22,33]])
>>> b = DimArray([[4,5,6],[44,55,66]])
>>> concatenate((a, b), axis=0)
dimarray: 12 non-null elements (0 null)
0 / x0 (4): 0 to 1
1 / x1 (3): 0 to 2
array([[ 1,  2,  3],
       [11, 22, 33],
       [ 4,  5,  6],
       [44, 55, 66]])
>>> concatenate((a, b), axis='x1')
dimarray: 12 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (6): 0 to 2
array([[ 1,  2,  3,  4,  5,  6],
       [11, 22, 33, 44, 55, 66]])

```

`dimarray.stack_ds` (*datasets*, *axis*, *keys=None*, *align=False*, ***kwargs*)
stack dataset along a new dimension

Parameters

datasets: sequence or dict of datasets

axis: str, new dimension along which to stack the dataset

keys, optional: stack axis values, useful if dataset is a sequence, or a non-ordered dictionary

align, optional: if True, align axes (via reindexing) **prior** to stacking

****kwargs** [optional key-word arguments passed to align, if align is True]

Returns

stacked dataset

See also:

`concatenate_ds`, `stack`, `sort_axis`

Examples

```

>>> a = DimArray([1,2,3], dims=('dima',))
>>> b = DimArray([11,22], dims=('dimb',))
>>> ds = Dataset(a=a,b=b) # dataset of 2 variables from an experiment
>>> ds2 = Dataset(a=a*2,b=b*2) # dataset of 2 variables from a second experiment
>>> stack_ds([ds, ds2], axis='stackdim', keys=['exp1','exp2'])
Dataset of 2 variables
0 / stackdim (2): 'exp1' to 'exp2'
1 / dima (3): 0 to 2
2 / dimb (2): 0 to 1
a: ('stackdim', 'dima')
b: ('stackdim', 'dimb')

```

`dimarray.concatenate_ds` (*datasets*, *axis=0*, *align=False*, ***kwargs*)
concatenate two datasets along an existing dimension

Parameters

datasets: sequence of datasets

axis: axis along which to concatenate

align, optional: if True, align secondary axes (via reindexing) prior to concatenating

****kwargs** [optional key-word arguments passed to align, if align is True]

Returns

joint Dataset along axis

NOTE: will raise an error if variables are there which do not contain the required dimension

See also:

`stack_ds`, `concatenate`, `sort_axis`

Examples

```
>>> a = da.zeros(axes=[list('abc')], dims=('x0',)) # 1-D DimArray
>>> b = da.zeros(axes=[list('abc'), [1,2]], dims=('x0','x1')) # 2-D DimArray
>>> ds = Dataset(a=a,b=b) # dataset of 2 variables from an experiment
>>> a2 = da.ones(axes=[list('def')], dims=('x0',))
>>> b2 = da.ones(axes=[list('def'), [1,2]], dims=('x0','x1')) # 2-D DimArray
>>> ds2 = Dataset(a=a2,b=b2) # dataset of 2 variables from a second experiment
>>> concatenate_ds([ds, ds2])
Dataset of 2 variables
0 / x0 (6): 'a' to 'f'
1 / x1 (2): 1 to 2
a: ('x0',)
b: ('x0', 'x1')
```

6.4.2 Align

`dimarray.align_axes` (**args*, ***kwargs*)
Deprecated. Now renamed to align

`dimarray.align_dims` (**arrays*)

Align dimensions of a list of arrays so that they are ready for broadcast.

Method: inserting singleton axes at the right place and transpose where needed. Note : not part of public API, but used in other dimarray modules

Examples

```

>>> import dimarray as da
>>> import numpy as np
>>> x = da.DimArray(np.arange(2), dims=('x0',))
>>> y = da.DimArray(np.arange(3), dims=('x1',))
>>> align_dims(x, y)
[dimarray: 2 non-null elements (0 null)
0 / x0 (2): 0 to 1
1 / x1 (1): None to None
array([[0],
       [1]]), dimarray: 3 non-null elements (0 null)
0 / x0 (1): None to None
1 / x1 (3): 0 to 2
array([[0, 1, 2]])]

```

`dimarray.broadcast_arrays(*arrays)`

Analogous to `numpy.broadcast_arrays`

but with looser requirements on input shape and returns copy instead of views

Parameters

arrays [variable list of DimArrays]

Returns

list of DimArrays

Examples

Just as `numpy`'s `broadcast_arrays`

```

>>> import dimarray as da
>>> x = da.DimArray([[1,2,3]])
>>> y = da.DimArray([[1],[2],[3]])
>>> da.broadcast_arrays(x, y)
[dimarray: 9 non-null elements (0 null)
0 / x0 (3): 0 to 2
1 / x1 (3): 0 to 2
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]]), dimarray: 9 non-null elements (0 null)
0 / x0 (3): 0 to 2
1 / x1 (3): 0 to 2
array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])]

```

6.4.3 Interpolate

`dimarray.interp2d(dim_array, newaxes, dims=(-2, -1), **kwargs)`

Two-dimensional interpolation

Parameters

dim_array [DimArray instance]

newaxes [sequence of two array-like, or dict.] axes on which to interpolate

dims [sequence of two axis names or integer rank, optional] Indicate dimensions which match *newaxes*. By default (-2, -1) (last two dimensions).

****kwargs** [passed to `scipy.interpolate.RegularGridInterpolator`] method : 'nearest' or 'linear' (default) bounds_error : True by default fill_value : np.nan by default, but set to None to extrapolate outside bounds.

Returns

dim_array_int [DimArray instance] interpolated array

Examples

```
>>> from dimarray import DimArray, interp2d
>>> x = np.array([0, 1, 2])
>>> y = np.array([0, 10])
>>> a = DimArray([[0,0,1],[1,0.,0.]], [('y',y), ('x',x)])
>>> a
dimarray: 6 non-null elements (0 null)
0 / y (2): 0 to 10
1 / x (3): 0 to 2
array([[0., 0., 1.],
       [1., 0., 0.]])
>>> newx = [0.5, 1.5]
>>> newy = np.linspace(0,10,5)
>>> ai = interp2d(a, [newy, newx])
>>> ai
dimarray: 10 non-null elements (0 null)
0 / y (5): 0.0 to 10.0
1 / x (2): 0.5 to 1.5
array([[0.   , 0.5  ],
       [0.125, 0.375],
       [0.25 , 0.25 ],
       [0.375, 0.125],
       [0.5  , 0.   ]])
```

Use `dims` keyword argument if new axes order does not match array dimensions >>> (ai == interp2d(a, [newx, newy], dims=('x','y'))).all() True

Out-of-bounds filled with NaN: >>> newx = [-1, 1] >>> newy = [-5, 0, 10] >>> interp2d(a, [newy, newx], bounds_error=False) dimarray: 2 non-null elements (4 null) 0 / y (3): -5 to 10 1 / x (2): -1 to 1 array([[nan, nan], [nan, 0.], [nan, 0.]])

Nearest neighbor interpolation and out-of-bounds extrapolation >>> interp2d(a, [newy, newx], method='nearest', bounds_error=False, fill_value=None) dimarray: 6 non-null elements (0 null) 0 / y (3): -5 to 10 1 / x (2): -1 to 1 array([[0., 0.], [0., 0.], [1., 0.]])

6.4.4 Stats

`dimarray.percentile(a, pct, axis=0, newaxis=None, out=None, overwrite_input=False)`
calculate percentile along an axis

Parameters

pct: float, percentile or sequence of percentiles ($0 < \text{pct} < 100$)

axis, optional, default 0: axis along which to compute percentiles

newaxis, optional: name of the new percentile axis, if more than one pct. By default, append “_percentile” to the axis name on which the transformation is applied.

out, overwrite_input: passed to numpy’s percentile method (see documentation)

Returns

ptiles: DimArray or scalar whose required axis has been reduced or replaced by percentiles

Examples

```
>>> from dimarray import DimArray
>>> np.random.seed(0) # for reproductibility of results
>>> a = DimArray(np.random.randn(1000), dims=['sample'])
>>> percentile(a, 50)
-0.058028034799627745
```

```
>>> percentile(a, [50, 95])
dimarray: 2 non-null elements (0 null)
0 / sample_percentile (2): 50 to 95
array([-0.05802803,  1.66012041])
```

6.4.5 Read netCDF data

`dimarray.read_nc(f, names=None, *args, **kwargs)`

Wrapper around DatasetOnDisk.read

Read one or several variables from one or several netCDF file

Parameters

f [str or netCDF handle] netCDF file to read from or regular expression

names [None or list or str, optional] variable name(s) to read default is None

indices [int or list or slice (single-dimensional indices)]

or a tuple of those (multi-dimensional) or *dict* of { axis name : axis indices }

Indices refer to Dataset axes. Any item that does not possess one of the dimensions will not be indexed along that dimension. For example, scalar items will be left unchanged whatever indices are provided.

indexing [{‘label’, ‘position’}, optional] Indexing mode. - “label”: indexing on axis labels (default) - “position”: use numpy-like position index Default value can be changed in `dimarray.rcParams[‘indexing.by’]`

tol [float, optional] tolerance when looking for numerical values, e.g. to use nearest neighbor search, default *None*.

keepdims [bool, optional] keep singleton dimensions (default False)

axis [str, optional] When reading multiple files, axis along which to join the dimarrays or datasets. If the axis already exist, the resulting arrays will be concatenated, otherwise they

will be stacked along a new array (in the sense of the numpy functions *concatenate* and *stack*)

keys [sequence, optional] When reading multiple files, keys for the join axis. If the axis already exists in the dataset, the concatenated dataset/dimarray will be re-indexed along the provided key, otherwise the keys will be used to create a new axis for stacking. In the latter case, keys' length needs to exactly match the number of input files, and if not provided, file names will be taken instead. Note you may manually rename the axes later, or use the *set_axis* method.

align [bool, optional] When reading multiple files, passed to *stack* (new axis) or *concatenate* (existing axis) to reindex all arrays onto common axes. (in *concatenate* mode, the concatenation axis is *not* re-indexed of course, only the secondary axes) Default to False.

****kwargs** [optional key-word arguments passed to align, if align is True] When reading multiple files, passed to *stack* (new axis) or This includes: *sort* (False by default) and *join* ('outer' by default)

Returns

obj [DimArray or Dataset] depending on whether a (single) variable name is passed as argument (names) or not

See also:

DatasetOnDisk.read, *stack*, *concatenate*, *stack_ds*, *concatenate_ds*, *align*, *DimArray.write_nc*, Dataset.write_nc

Examples

```
>>> import os
>>> from dimarray import read_nc, get_datadir
```

Single netCDF file

```
>>> ncfile = os.path.join(get_datadir(), 'cmip5.CSIRO-Mk3-6-0.nc')
```

```
>>> data = read_nc(ncfile) # load full file
>>> data
Dataset of 2 variables
0 / time (451): 1850 to 2300
1 / scenario (5): u'historical' to u'rcp85'
ts1: (u'time', u'scenario')
temp: (u'time', u'scenario')
>>> data = read_nc(ncfile, 'temp') # only one variable
>>> data = read_nc(ncfile, 'temp', indices={"time": slice(2000, 2100), "scenario":
↳ "rcp45"}) # load only a chunk of the data
>>> data = read_nc(ncfile, 'temp', indices={"time": 1950.3}, tol=0.5) #
↳ approximate matching, adjust tolerance
>>> data = read_nc(ncfile, 'temp', indices={"time": -1}, indexing='position') #
↳ integer position indexing
```

Multiple files Read variable 'temp' across multiple files (representing various climate models) In this case the variable is a time series, whose length may vary across experiments (thus align=True is passed to reindex axes before stacking)

```
>>> direc = get_datadir()
>>> temp = da.read_nc(direc+'cmip5.*.nc', 'temp', align=True, axis='model')
```

A new ‘model’ axis is created labeled with file names. It is then possible to rename it more appropriately, e.g. keeping only the part directly relevant to identify the experiment:

```
>>> getmodel = lambda x: os.path.basename(x).split('.')[1] # extract model name_
↳from path
>>> temp.set_axis(getmodel, axis='model') # would return a copy if inplace is not_
↳specified
>>> temp
dimarray: 9114 non-null elements (6671 null)
0 / model (7): 'CSIRO-Mk3-6-0' to 'MPI-ESM-MR'
1 / time (451): 1850 to 2300
2 / scenario (5): u'historical' to u'rcp85'
array(...)
```

This works on datasets as well:

```
>>> ds = da.read_nc(direct+'cmip5.*.nc', align=True, axis='model')
>>> ds.set_axis(getmodel, axis='model')
>>> ds
Dataset of 2 variables
0 / model (7): 'CSIRO-Mk3-6-0' to 'MPI-ESM-MR'
1 / time (451): 1850 to 2300
2 / scenario (5): u'historical' to u'rcp85'
ts1: ('model', u'time', u'scenario')
temp: ('model', u'time', u'scenario')
```

`dimarray.summary_nc` (*fname*, *name=None*, *metadata=False*)

Print summary information about the content of a netCDF file *Deprecated*, see `dimarray.open_nc`

`dimarray.get_datadir` ()

Return directory name for the datasets

`dimarray.get_ncfile` (*fname='cmip5.CSIRO-Mk3-6-0.nc'*)

Return one netCDF file

6.4.6 dimarray options

`dimarray.print_options` ()

`dimarray.get_option` (*name*)

`dimarray.set_option` (*name*, *value*)

set global options

6.5 Table numpy vs dimarray

Table of correspondence between numpy ndarray and dimarray's functions and methods

array creation		
numpy	dimarray	comments
array	DimArray	In dimarray need to provide axes information in addition to values.
•	DimArray.from_kw	Same as DimArray() but provide axes as key-words.
•	array	same as DimArray()
•	array_kw	same as DimArray.from_kw()
zeros	zeros	These functions are similar to numpy except that they require <i>axes</i> parameter, or a <i>shape=</i> parameter for automatic labelling.
ones	ones	
empty	empty	
zeros_like	zeros_like	
ones_like	ones_like	
empty_like	empty_like	

Note: The *array* and *array_kw* forms (to be used as `da.array()`) are attempts to make the array definition less verbose. They are experimental and may change in the future.

reshaping		
numpy	dimarray	comments
a.T	a.T	Transpose a 2-dimensional array
a.transpose()	a.transpose()	Transpose or permute array. In dimarray also accept axis names
a.swapaxes()	a.swapaxes()	Swap two axes. In dimarray also accept axis names. e.g. a.swapaxes('time', 0) to bring the 'time' dimension as first axis to ease indexing.
a.reshape()	a.reshape()	Change array shape without changing the size. There are a few differences in dimarray compared to numpy: - a dimension cannot be broken down (e.g. 4 => 2x2) - the full shape of the array is given via axis names e.g. a.reshape('time,percentile','scenario') will flatten (<i>group</i>) the dimensions <i>time</i> and <i>percentile</i> to end up with a 2-D array, and transpose the array as necessary to get to the desired shape. If only transposing (permutation) is needed, the use of <i>transpose</i> is preferred for clarity.
•	a.group()	Flatten two axes into one: it is for <i>reshape</i> what <i>swapaxes</i> is to <i>transpose</i> .
•	a.ungroup()	Inflate two or more "grouped" axes (undo a.group()).
a.flatten()	a.flatten()	Flatten array. In dimarray the axes are transformed into tuples (<i>GroupedAxis</i>).
a[np.newaxis]	a.newaxis()	In numpy, add a singleton dimension, useful for broadcasting in an operation. In dimarray, broadcasting is based on dimension names and therefore streamlined without the need to provide this extra information, make this option less relevant in the public API. In dimarray this is a method since it requires the name of the new axis, and by extension, if the new axis' values are also provided it can also combine functionality of <i>repeat</i> .
a.squeeze()	a.squeeze()	idem, but also accept axis names (opposite of <i>newaxis</i>)
a.repeat()	a.repeat()	In dimarray it's mostly an internal method that only works on singleton dimensions. This is of no much practical use. Use <i>newaxis</i> instead.
broadcast()	a.broadcast()	Dimarray's method similar to numpy's function. Add or remove singleton axes to make it match another array's dimensions, but without repeating (so that the shapes do not necessarily match, but it is ready for binary operations
6.5. Table numpy vs dimarray		71

Note: The names *group* and *ungroup* may be confusing and could change in the future (e.g. to flatten and inflate, or unflatten)

The methods below are mostly similar across the packages, but dimarray also accepts axis name instead of axis rank as *axis=*. An optional *skipna=* parameter can be provided to ignore nans (default to *False*). Note also that in many cases, when a *tuple* of axis names is provided the array is first partially flattened (grouped axis) before the dimension is reduced

reduce, accumulate (along-axis transformation)		
numpy	dimarray	comments
a.max()	a.max()	
a.min()	a.min()	
a.ptp()	a.ptp()	
a.median()	a.median()	
a.all()	a.all()	
a.any()	a.any()	
a.prod()	a.prod()	
a.sum()	a.sum()	
a.mean()	a.mean()	
a.std()	a.std()	
a.var()	a.var()	
a.argmax()	a.argmax()	in dimarray, returns axis value of max instead of integer position on the axis
a.argmin()	a.argmin()	idem
a.cumsum()	a.cumsum()	
a.cumprod()	a.cumprod()	
diff(a,...)	a.diff()	as method, and with <i>scheme=</i> parameter (“forward”, “centered”, “backward”)

Maintainance of the documentation

The Documentation is generated with Sphinx from ReStructuredTxt files (‘.rst’). Many sections however also exist and are maintained as notebooks, using basic formatting. The conversion from notebook to rst is done via a script (take a look) and has been included as a Makefile command (make rst).

The workflow is as follow:

```
1. cd docs
2. ... # edit notebooks in notebooks/
3. ... # edit rst files
4. make rst # convert every notebook in docs/notebooks to rst in docs/_notebooks_rst
5. make html # this could also be combine above in make rst html
6. ... # check the result in docs/_build/html/index.html
7. ... # iterate until you are happy with the result
8. git add / rm / ci # commit the change
9. git push # push to github
```

Pushing to github will update the doc at readthedocs automatically.

Note: Step 4 will work only on unix system because bash is involved in one of the scripts (this could actually be written in python easily) There might also be other dependencies involved, maybe even the ipython version (did that with the latest 3.0.0).

Note: readthedocs will re-compile the rst files to html, so that steps 5-6 using your local sphinx installation are only for you to check the results before pushing.

Note: To compile locally with sphinx, you need to download sphinx of course, but also numpydoc (which parse numpy-like docstrings) e.g. “pip -r docs/readthedocs-pip-requirements.txt”

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`

Symbols

`__getitem__()` (dimarray.DimArray method), 50
`__init__()` (dimarray.DimArray method), 34

A

`align_axes()` (in module dimarray), 64
`align_dims()` (in module dimarray), 64
`all()` (dimarray.DimArray method), 44
`any()` (dimarray.DimArray method), 44
`argmax()` (dimarray.DimArray method), 48
`argmin()` (dimarray.DimArray method), 48

B

`box()` (dimarray.DimArray method), 50
`broadcast()` (dimarray.DimArray method), 41
`broadcast_arrays()` (in module dimarray), 65

C

`concatenate()` (in module dimarray), 62
`concatenate_ds()` (in module dimarray), 63
`contour()` (dimarray.DimArray method), 60
`contourf()` (dimarray.DimArray method), 60
`cumprod()` (dimarray.DimArray method), 49
`cumsum()` (dimarray.DimArray method), 49

D

`diff()` (dimarray.DimArray method), 49
`dimarray.Axes` (built-in class), 61
`dimarray.Axis` (built-in class), 61
`dimarray.GroupedAxis` (built-in class), 61
`dropna()` (dimarray.DimArray method), 56

F

`fillna()` (dimarray.DimArray method), 57
`flatten()` (dimarray.DimArray method), 37
`from_pandas()` (dimarray.DimArray class method), 58

G

`get_datadir()` (in module dimarray), 69

`get_ncfile()` (in module dimarray), 69
`get_option()` (in module dimarray), 69

I

`interp2d()` (in module dimarray), 65
`ix()` (dimarray.DimArray method), 50

M

`max()` (dimarray.DimArray method), 41
`mean()` (dimarray.DimArray method), 46
`median()` (dimarray.DimArray method), 43
`min()` (dimarray.DimArray method), 42

P

`pcolor()` (dimarray.DimArray method), 60
`percentile()` (in module dimarray), 66
`plot()` (dimarray.DimArray method), 59
`print_options()` (in module dimarray), 69
`prod()` (dimarray.DimArray method), 45
`ptp()` (dimarray.DimArray method), 43
`put()` (dimarray.DimArray method), 53

R

`read_nc()` (in module dimarray), 67
`reindex_axis()` (dimarray.DimArray method), 54
`reindex_like()` (dimarray.DimArray method), 54
`repeat()` (dimarray.DimArray method), 40
`reset_axis()` (dimarray.DimArray method), 54
`reshape()` (dimarray.DimArray method), 36

S

`set_option()` (in module dimarray), 69
`setna()` (dimarray.DimArray method), 57
`sort_axis()` (dimarray.DimArray method), 55
`squeeze()` (dimarray.DimArray method), 39
`stack()` (in module dimarray), 61
`stack_ds()` (in module dimarray), 63
`std()` (dimarray.DimArray method), 47
`sum()` (dimarray.DimArray method), 46

`summary_nc()` (in module `dimarray`), [69](#)
`swapaxes()` (`dimarray.DimArray` method), [36](#)

T

`take()` (`dimarray.DimArray` method), [50](#)
`to_dataset()` (`dimarray.DimArray` method), [59](#)
`to_larray()` (`dimarray.DimArray` method), [59](#)
`to_pandas()` (`dimarray.DimArray` method), [58](#)
`transpose()` (`dimarray.DimArray` method), [35](#)

U

`unflatten()` (`dimarray.DimArray` method), [39](#)

V

`var()` (`dimarray.DimArray` method), [48](#)

W

`write_nc()` (`dimarray.DimArray` method), [59](#)