
LArray Documentation

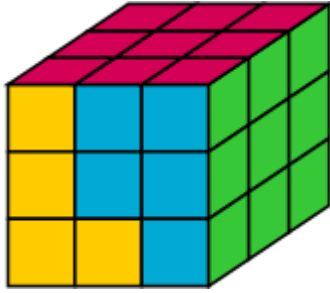
Release 0.32

Gaëtan de Menten, Geert Bryon, Johan Duyck, Alix Damman

Apr 04, 2020

CONTENTS

1	Library Highlights	3
2	Documentation	5
3	Get in touch	7
4	Contents	9
4.1	Installation	9
4.2	Tutorial	11
4.3	API Reference	135
5	Indices and tables	463
6	Appendix	465
6.1	Change log	465
6.2	How to contribute	566
	Bibliography	573
	Index	575

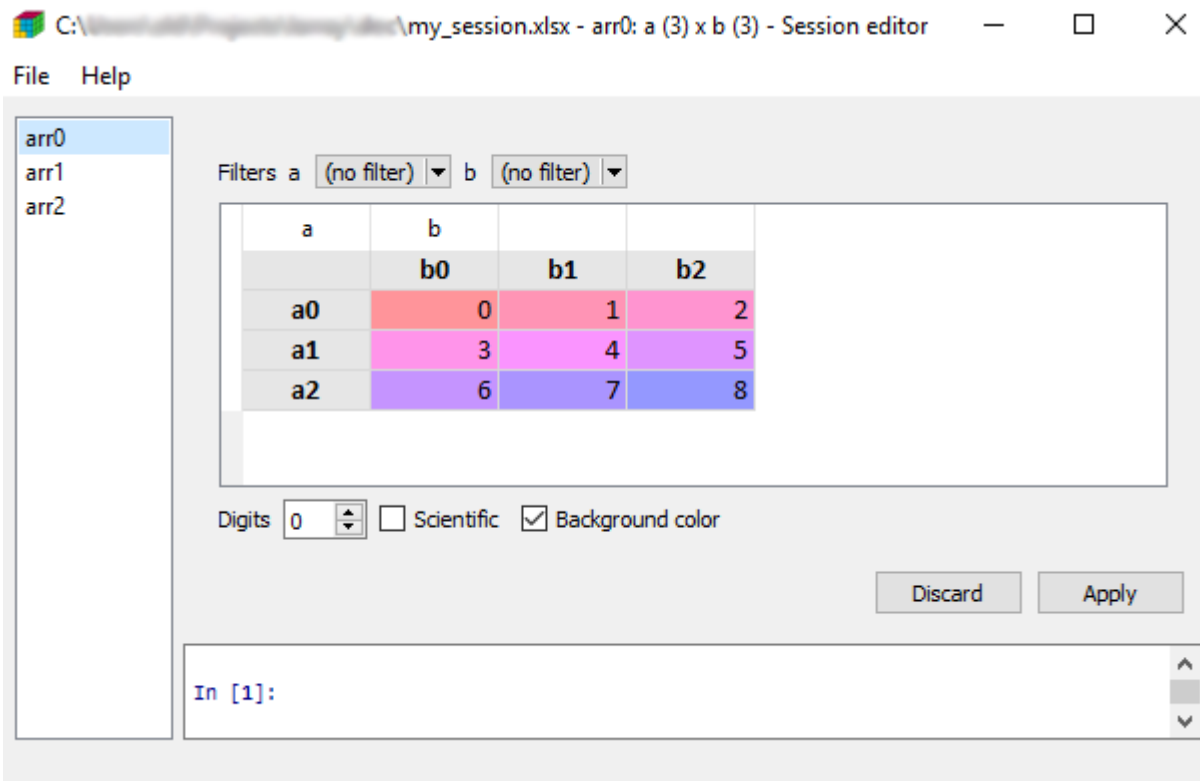


LArray

LArray is open source Python library that aims to provide tools for easy exploration and manipulation of N-dimensional labelled data structures.

LIBRARY HIGHLIGHTS

- N-dimensional labelled array objects to store and manipulate multi-dimensional data
- I/O functions for reading and writing arrays in different formats: CSV, Microsoft Excel, HDF5, pickle
- Arrays can be grouped into Session objects and loaded/dumped at once
- User interface with an IPython console for rapid exploration of data
- Compatible with the pandas library: Array objects can be converted into pandas DataFrame and vice versa.



DOCUMENTATION

The official documentation is hosted on ReadTheDocs at <http://larray.readthedocs.io/en/stable/>

GET IN TOUCH

- To be informed of each new release, please subscribe to the announce [mailing list](#).
- For questions, ideas or general discussion, please use the [Google Users Group](#).
- To report bugs, suggest features or view the source code, please go to our [GitHub website](#).

CONTENTS

4.1 Installation

4.1.1 Pre-built binaries

The easiest route to installing larray is through [Conda](#). For all platforms installing larray can be done with:

```
conda install -c larray-project larray
```

This will install a lightweight version of larray depending only on Numpy and Pandas libraries only. Additional libraries are required to use the included graphical user interface, make plots or use special I/O functions for easy dump/load from Excel or HDF files. Optional dependencies are described below.

Installing larray with all optional dependencies can be done with

```
conda install -c larray-project larrayenv
```

You can also first add the channel *larray-project* to your channel list

```
conda config --add channels larray-project
```

and then install larray (or larrayenv) as

```
conda install larray
```

4.1.2 Building from source

The latest release of LArray is available from <https://github.com/larray-project/larray.git>

Once you have satisfied the requirements detailed below, simply run:

```
python setup.py install
```

4.1.3 Required Dependencies

- Python 2.7, 3.6 or 3.7
- `numpy` (1.13 or later)
- `pandas` (0.20 or later)

4.1.4 Optional Dependencies

For IO (HDF, Excel)

- `pytables`: for working with files in HDF5 format.
- `xlwings`: recommended package to get benefit of all Excel features of LArray. Only available on Windows and Mac platforms.
- `xlrd`: for reading data and formatting information from older Excel files (ie: `.xls`)
- `openpyxl`: recommended package for reading and writing Excel 2010 files (ie: `.xlsx`)
- `xlsxwriter`: alternative package for writing data, formatting information and, in particular, charts in the Excel 2010 format (ie: `.xlsx`)
- `larray_eurostat`: provides functions to easily download EUROSTAT files as larray objects. Currently limited to TSV files.

For Graphical User Interface

LArray includes a graphical user interface to view, edit and compare arrays.

- `pyqt` (4 or 5): required by *larray-editor* (see below).
- `pyside`: alternative to PyQt.
- `qtpy`: required by *larray-editor*. Provides support for PyQt5, PyQt4 and PySide using the PyQt5 layout.
- `larray-editor`: required to use the graphical user interface associated with larray. It assumes that *qtpy* and *pyqt* or *pyside* are installed. On windows, creates also a menu `LArray` in the Windows Start Menu.

For plotting

- `matplotlib`: required for plotting.

4.1.5 Update

If larray has been installed using conda, update is done via

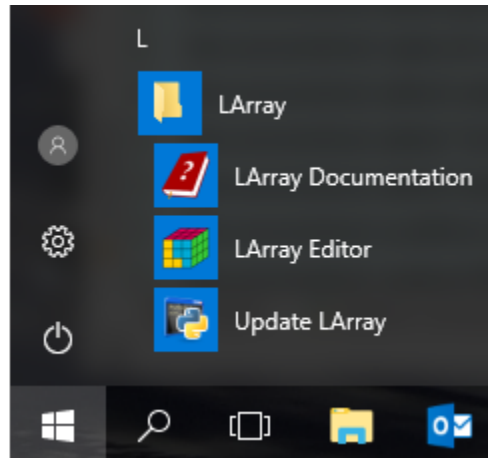
```
conda update larray
```

Be careful if you have installed optional dependencies. In that case, you may have to update some of them.

If larray has been installed using conda via `larrayenv`, you simply must do

```
conda update larrayenv
```

For Windows users who have larrayenv (≥ 0.25) installed, simply click on the `Update LArray` link in the the Windows Start Menu > LArray.



4.2 Tutorial

This is an overview of the LArray library. It is not intended to be a fully comprehensive manual. It is mainly dedicated to help new users to familiarize with it and others to remind essentials.

4.2.1 Getting Started

The purpose of the present **Getting Started** section is to give a quick overview of the main objects and features of the LArray library. To get a more detailed presentation of all capabilities of LArray, read the next sections of the tutorial.

The *API Reference* section of the documentation give you the list of all objects, methods and functions with their individual documentation and examples.

To use the LArray library, the first thing to do is to import it:

```
[1]: from larray import *
```

To know the version of the LArray library installed on your machine, type:

```
[2]: from larray import __version__
__version__
```

```
[2]: '0.32'
```

Create an array

Working with the LArray library mainly consists of manipulating *Array* data structures. They represent N-dimensional labelled arrays and are composed of raw data (NumPy ndarray), axes and optionally some metadata.

An *Axis* object represents a dimension of an array. It contains a list of labels and has a name:

```
[3]: # define some axes to be used later
age = Axis(['0-9', '10-17', '18-66', '67+'], 'age')
gender = Axis(['female', 'male'], 'gender')
time = Axis([2015, 2016, 2017], 'time')
```

The labels allow to select subsets and to manipulate the data without working with the positions of array elements directly.

To create an array from scratch, you need to supply data and axes:

```
[4]: # define some data. This is the belgian population (in thousands). Source: eurostat.
data = [[633, 635, 634],
        [663, 665, 664]],
        [[484, 486, 491],
        [505, 511, 516]],
        [[3572, 3581, 3583],
        [3600, 3618, 3616]],
        [[1023, 1038, 1053],
        [756, 775, 793]]

# create an Array object
pop = Array(data, axes=[age, gender, time])
pop
```

```
[4]:  age  gender\time  2015  2016  2017
      0-9      female   633    635    634
      0-9      male    663    665    664
     10-17     female   484    486    491
     10-17     male    505    511    516
     18-66     female  3572   3581   3583
     18-66     male   3600   3618   3616
      67+     female  1023   1038   1053
      67+     male    756    775    793
```

You can optionally attach some metadata to an array:

```
[5]: # attach some metadata to the pop array
pop.meta.title = 'population by age, sex and year'
pop.meta.source = 'Eurostat'

# display metadata
pop.meta
```

```
[5]: title: population by age, sex and year
      source: Eurostat
```

To get a short summary of an array, type:

```
[6]: # Array summary: metadata + dimensions + description of axes
pop.info
```

```
[6]: title: population by age, sex and year
      source: Eurostat
      4 x 2 x 3
      age [4]: '0-9' '10-17' '18-66' '67+'
      gender [2]: 'female' 'male'
      time [3]: 2015 2016 2017
      dtype: int64
      memory used: 192 bytes
```


Create an array filled with predefined values

Arrays filled with predefined values can be generated through *dedicated functions*:

- `zeros` : creates an array filled with 0
- `ones` : creates an array filled with 1
- `full` : creates an array filled with a given value
- `sequence` : creates an array by sequentially applying modifications to the array along axis.
- `ndtest` : creates a test array with increasing numbers as data

```
[7]: zeros([age, gender])
```

```
[7]: age\gender  female  male
      0-9        0.0    0.0
      10-17      0.0    0.0
      18-66      0.0    0.0
      67+       0.0    0.0
```

```
[8]: ones([age, gender])
```

```
[8]: age\gender  female  male
      0-9        1.0    1.0
      10-17      1.0    1.0
      18-66      1.0    1.0
      67+       1.0    1.0
```

```
[9]: full([age, gender], fill_value=10.0)
```

```
[9]: age\gender  female  male
      0-9       10.0   10.0
      10-17     10.0   10.0
      18-66     10.0   10.0
      67+      10.0   10.0
```

```
[10]: sequence(age)
```

```
[10]: age  0-9  10-17  18-66  67+
        0     1     2     3
```

```
[11]: ndtest([age, gender])
```

```
[11]: age\gender  female  male
      0-9         0       1
      10-17       2       3
      18-66       4       5
      67+        6       7
```

Save/Load an array

The LArray library offers many I/O functions to read and write arrays in various formats (CSV, Excel, HDF5). For example, to save an array in a CSV file, call the method `to_csv`:

```
[12]: # save our pop array to a CSV file
pop.to_csv('belgium_pop.csv')
```

The content of the CSV file is then:

```
age,gender\time,2015,2016,2017
0-9,female,633,635,634
0-9,male,663,665,664
10-17,female,484,486,491
10-17,male,505,511,516
18-66,female,3572,3581,3583
18-66,male,3600,3618,3616
67+,female,1023,1038,1053
67+,male,756,775,793
```

Note: In CSV or Excel files, the last dimension is horizontal and the names of the last two dimensions are separated by a backslash .

To load a saved array, call the function `read_csv`:

```
[13]: pop = read_csv('belgium_pop.csv')
pop
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-13-87e83451a034> in <module>
----> 1 pop = read_csv('belgium_pop.csv')
      2 pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
      700         else:
      701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
      703     return wrapper
      704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
      231         raw = False
      232
--> 233     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)
      234
      235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
```

(continues on next page)

(continued from previous page)

```

338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
236     if cartesian_prod:
237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238             fill_value=fill_value,
↳ **kwargs)
239     else:
240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
57     if isinstance(idx, pd.core.index.MultiIndex):
58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

Other input/output functions are described in the *Input/Output* section of the API documentation.

Selecting a subset

To select an element or a subset of an array, use brackets []. In Python we usually use the term *indexing* for this operation.

Let us start by selecting a single element:

```

[14]: pop['67+', 'female', 2017]
[14]: 1053

```

Labels can be given in arbitrary order:

```
[15]: pop[2017, 'female', '67+']
[15]: 1053
```

When selecting a larger subset the result is an array:

```
[16]: pop['female']
[16]: age\time  2015  2016  2017
      0-9      633   635   634
      10-17    484   486   491
      18-66   3572  3581  3583
      67+    1023  1038  1053
```

When selecting several labels for the same axis, they must be given as a list (enclosed by [])

```
[17]: pop['female', ['0-9', '10-17']]
[17]: age\time  2015  2016  2017
      0-9      633   635   634
      10-17    484   486   491
```

You can also select *slices*, which are all labels between two bounds (we usually call them the start and stop bounds). Specifying the start and stop bounds of a slice is optional: when not given, start is the first label of the corresponding axis, stop the last one:

```
[18]: # in this case '10-17':'67+' is equivalent to ['10-17', '18-66', '67+']
      pop['female', '10-17':'67+']
[18]: age\time  2015  2016  2017
      10-17    484   486   491
      18-66   3572  3581  3583
      67+    1023  1038  1053
```

```
[19]: # : '18-66' selects all labels between the first one and '18-66'
      # 2017: selects all labels between 2017 and the last one
      pop[:, '18-66', 2017:]
[19]: age  gender\time  2017
      0-9      female  634
      0-9      male   664
      10-17     female  491
      10-17     male   516
      18-66     female  3583
      18-66     male   3616
```

Note: Contrary to slices on normal Python lists, the stop bound is included in the selection.

Selecting by labels as above only works as long as there is no ambiguity. When several axes have some labels in common and you do not specify explicitly on which axis to work, it fails with an error ending with something like `ValueError: is ambiguous (valid in ,)`.

For example, imagine you need to work with an ‘immigration’ array containing two axes sharing some common labels:

```
[20]: country = Axis(['Belgium', 'Netherlands', 'Germany'], 'country')
      citizenship = Axis(['Belgium', 'Netherlands', 'Germany'], 'citizenship')

      immigration = ndtest((country, citizenship, time))

      immigration
```

```
[20]:
```

country	citizenship\time	2015	2016	2017
Belgium	Belgium	0	1	2
Belgium	Netherlands	3	4	5
Belgium	Germany	6	7	8
Netherlands	Belgium	9	10	11
Netherlands	Netherlands	12	13	14
Netherlands	Germany	15	16	17
Germany	Belgium	18	19	20
Germany	Netherlands	21	22	23
Germany	Germany	24	25	26

If we try to get the number of Belgians living in the Netherlands for the year 2017, we might try something like:

```
immigration['Netherlands', 'Belgium', 2017]
```

... but we receive back a volley of insults:

```
[some long error message ending with the line below]
[...]
ValueError: Netherlands is ambiguous (valid in country, citizenship)
```

In that case, we have to specify explicitly which axes the ‘Netherlands’ and ‘Belgium’ labels we want to select belong to:

```
[21]: immigration[country['Netherlands'], citizenship['Belgium'], 2017]
```

```
[21]: 11
```

Aggregation

The LArray library includes many *aggregations methods*: sum, mean, min, max, std, var, ...

For example, assuming we still have an array in the pop variable:

```
[22]: pop
```

```
[22]:
```

age	gender\time	2015	2016	2017
0-9	female	633	635	634
0-9	male	663	665	664
10-17	female	484	486	491
10-17	male	505	511	516
18-66	female	3572	3581	3583
18-66	male	3600	3618	3616
67+	female	1023	1038	1053
67+	male	756	775	793

We can sum along the ‘sex’ axis using:

```
[23]: pop.sum(gender)
```

```
[23]: age\time  2015  2016  2017
      0-9    1296  1300  1298
      10-17   989   997  1007
      18-66  7172  7199  7199
      67+   1779  1813  1846
```

Or sum along both ‘age’ and ‘sex’:

```
[24]: pop.sum(age, gender)
```

```
[24]: time    2015    2016    2017
      11236   11309   11350
```

It is sometimes more convenient to aggregate along all axes **except** some. In that case, use the aggregation methods ending with `_by`. For example:

```
[25]: pop.sum_by(time)
```

```
[25]: time    2015    2016    2017
      11236   11309   11350
```

Groups

A *Group* object represents a subset of labels or positions of an axis:

```
[26]: children = age['0-9', '10-17']
      children
```

```
[26]: age['0-9', '10-17']
```

It is often useful to attach them an explicit name using the `>>` operator:

```
[27]: working = age['18-66'] >> 'working'
      working
```

```
[27]: age['18-66'] >> 'working'
```

```
[28]: nonworking = age['0-9', '10-17', '67+'] >> 'nonworking'
      nonworking
```

```
[28]: age['0-9', '10-17', '67+'] >> 'nonworking'
```

Still using the same `pop` array:

```
[29]: pop
```

```
[29]:  age  gender\time  2015  2016  2017
      0-9      female   633   635   634
      0-9      male    663   665   664
      10-17     female   484   486   491
      10-17     male    505   511   516
      18-66     female  3572  3581  3583
      18-66     male   3600  3618  3616
      67+      female  1023  1038  1053
      67+      male    756   775   793
```

Groups can be used in selections:

```
[30]: pop[working]
[30]: gender\time  2015  2016  2017
      female  3572  3581  3583
      male   3600  3618  3616
```

```
[31]: pop[nonworking]
[31]:  age  gender\time  2015  2016  2017
      0-9    female    633    635    634
      0-9    male     663    665    664
      10-17   female    484    486    491
      10-17   male     505    511    516
      67+    female   1023   1038   1053
      67+    male     756    775    793
```

or aggregations:

```
[32]: pop.sum(nonworking)
[32]: gender\time  2015  2016  2017
      female  2140  2159  2178
      male   1924  1951  1973
```

When aggregating several groups, the names we set above using `>>` determines the label on the aggregated axis. Since we did not give a name for the children group, the resulting label is generated automatically :

```
[33]: pop.sum((children, working, nonworking))
[33]:  age  gender\time  2015  2016  2017
      0-9,10-17   female  1117  1121  1125
      0-9,10-17   male   1168  1176  1180
      working     female  3572  3581  3583
      working     male   3600  3618  3616
      nonworking   female  2140  2159  2178
      nonworking   male   1924  1951  1973
```

Grouping arrays in a Session

Arrays may be grouped in *Session* objects. A session is an ordered dict-like container of Array objects with special I/O methods. To create a session, you need to pass a list of pairs (array_name, array):

```
[34]: pop = zeros([age, gender, time])
      births = zeros([age, gender, time])
      deaths = zeros([age, gender, time])

      # create a session containing the three arrays 'pop', 'births' and 'deaths'
      demo = Session(pop=pop, births=births, deaths=deaths)

      # displays names of arrays contained in the session
      demo.names
      # get an array
      demo['pop']
      # add/modify an array
      demo['foreigners'] = zeros([age, gender, time])
```

If you are using a Python version prior to 3.6, you will have to pass a list of pairs to the Session constructor otherwise the arrays will be stored in an arbitrary order in the new session. For example, the session above must be created using the syntax:

```
demo=Session([('pop', pop), ('births', births), ('deaths', deaths)]).
```

One of the main interests of using sessions is to save and load many arrays at once:

```
[35]: # dump all arrays contained in the session 'demo' in one HDF5 file
demo.save('demo.h5')
# load all arrays saved in the HDF5 file 'demo.h5' and store them in the session 'demo'
demo = Session('demo.h5')
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-35-bd6f192707b4> in <module>
      2 demo.save('demo.h5')
      3 # load all arrays saved in the HDF5 file 'demo.h5' and store them in the_
    ↪ session 'demo'
----> 4 demo = Session('demo.h5')

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
    --> 96             self.load(a0)
      97         else:
      98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
    ↪ display, **kwargs)
      426         else:
      427             handler = handler_cls(fname)
    --> 428             metadata, objects = handler.read(names, display=display, **kwargs)
      429             for k, v in objects.items():
      430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
      128             print("loading", type, "object", key, "...", end=' ')
      129             try:
    --> 130                 res[key] = self._read_item(key, type, *args, **kwargs)
      131             except Exception:
      132                 if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
    ↪ **kwargs)
      137         else:
      138             raise TypeError()
    --> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
      140
      141         def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill
    ↪ value, na, sort_rows, sort_columns, name, **kwargs)
```

(continues on next page)

(continued from previous page)

```

81         cartesian_prod = writer != 'LArray'
82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳columns, fill_value=fill_value,
--> 83             parse_header=False, cartesian_prod=cartesian_
↳prod)
84         if _meta is not None:
85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳**kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used
↳when cartesian_prod is set to False. '
242             'Please call the method sort_axes on the
↳returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to
↳convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idxs, sort)
41         Returns unique labels for each dimension.
42         """
--> 43         if isinstance(idxs, pd.core.index.MultiIndex):
44             if sort:
45                 return list(idxs.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

Graphical User Interface (viewer)

The LArray project provides an optional package called *larray-editor* allowing users to explore and edit arrays through a graphical interface.

The larray-editor tool is automatically available when installing the **larrayenv** metapackage from conda.

To explore the content of arrays in read-only mode, call the view function:

```

# shows the arrays of a given session in a graphical user interface
view(demo)

```

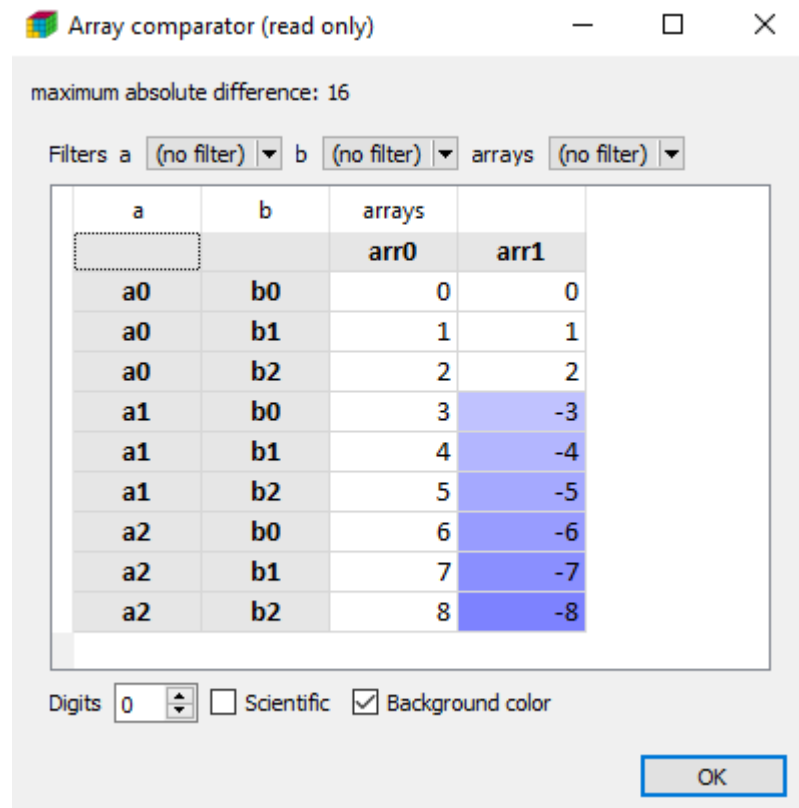
(continues on next page)

(continued from previous page)

```
# the session may be directly loaded from a file
view('demo.h5')

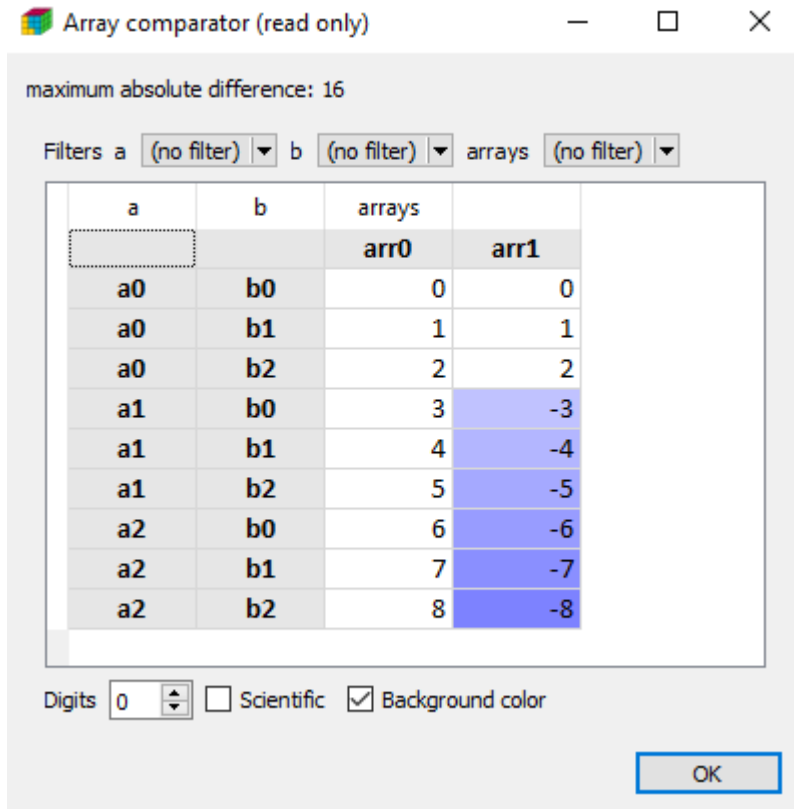
# creates a session with all existing arrays from the current namespace
# and shows its content
view()
```

To open the user interface in edit mode, call the `edit` function instead.



Finally, you can also visually compare two arrays or sessions using the `compare` function:

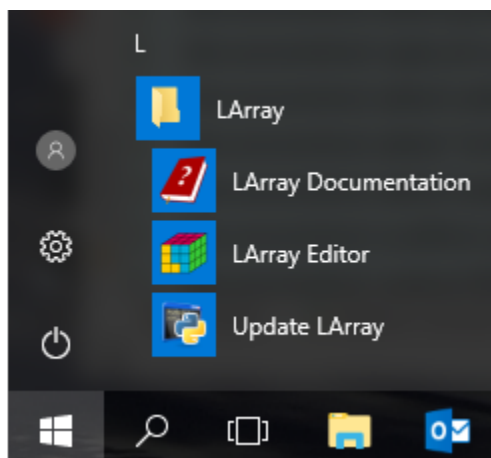
```
arr0 = ndtest((3, 3))
arr1 = ndtest((3, 3))
arr1[['a1', 'a2']] = -arr1[['a1', 'a2']]
compare(arr0, arr1)
```

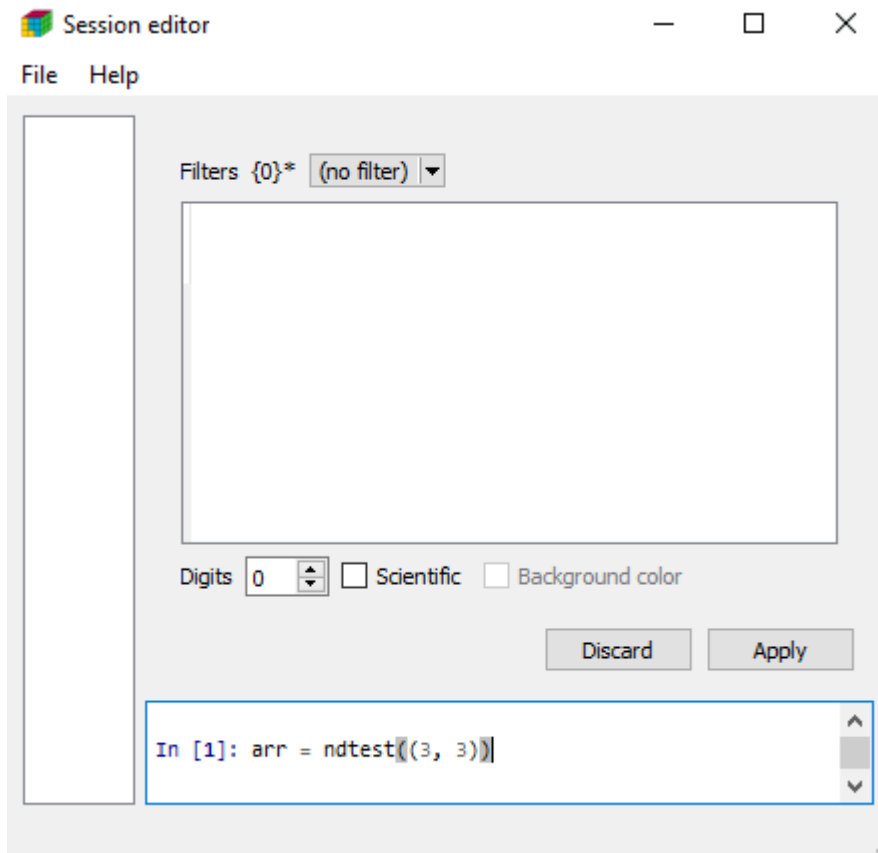


For Windows Users

Installing the `larray-editor` package on Windows will create a LArray menu in the Windows Start Menu. This menu contains:

- a shortcut to open the documentation of the last stable version of the library
- a shortcut to open the graphical interface in edit mode.
- a shortcut to update `larrayenv`.





Once the graphical interface is open, all LArray objects and functions are directly accessible. No need to start by `from larray import *`.

4.2.2 Presenting LArray objects (Axis, Groups, Array, Session)

Import the LArray library:

```
[1]: from larray import *
```

Axis

An `Axis` represents a dimension of an `Array` object. It consists of a name and a list of labels.

There are several ways to create an axis:

```
[2]: # create a wildcard axis
age = Axis(3, 'age')
# labels given as a list
time = Axis([2007, 2008, 2009], 'time')
# create an axis using one string
sex = Axis('sex=M,F')
# labels generated using a special syntax
other = Axis('other=A01..C03')

age, sex, time, other
```

```
[2]: (Axis(3, 'age'),
      Axis(['M', 'F'], 'sex'),
      Axis([2007, 2008, 2009], 'time'),
      Axis(['A01', 'A02', 'A03', 'B01', 'B02', 'B03', 'C01', 'C02', 'C03'], 'other'))
```

See the [Axis](#) section of the API Reference to explore all methods of Axis objects.

Groups

A Group represents a selection of labels from an Axis. It can optionally have a name (using operator >>). Groups can be used when selecting a subset of an array and in aggregations.

Group objects are created as follow:

```
[3]: # define an Axis object 'age'
age = Axis('age=0..100')

# create an anonymous Group object 'teens'
teens = age[10:20]
# create a Group object 'pensioners' with a name
pensioners = age[67:] >> 'pensioners'

teens

[3]: age[10:20]
```

It is possible to set a name or to rename a group after its declaration:

```
[4]: # method 'named' returns a new group with the given name
teens = teens.named('teens')

# operator >> is just a shortcut for the call of the method named
teens = teens >> 'teens'

teens

[4]: age[10:20] >> 'teens'
```

See the [Group](#) section of the API Reference to explore all methods of Group objects.

Array

An Array object represents a multidimensional array with labeled axes.

Create an array from scratch

To create an array from scratch, you need to provide the data and a list of axes. Optionally, metadata (title, description, creation date, authors, ...) can be associated to the array:

```
[5]: import numpy as np

# list of the axes
axes = [age, sex, time, other]
# data (the shape of data array must match axes lengths)
data = np.random.randint(100, size=[len(axis) for axis in axes])
```

(continues on next page)

(continued from previous page)

```
# metadata
meta = [('title', 'random array')]

arr = Array(data, axes, meta=meta)
arr
```

```
[5]: age  sex  time\other  A01  A02  A03  B01  B02  B03  C01  C02  C03
      0    M      2007    25    84    80    6    27    33    72    80    50
      0    M      2008    76    47    19    13    1    1    84    80    8
      0    M      2009    76    58    53    52    8    76    92    89    28
      0    F      2007    92    34    59    40    53    23    10    39    68
      0    F      2008    22    45    67    96    95    7    23    3    31
      ...  ...      ...    ...    ...    ...    ...    ...    ...    ...    ...
    100    M      2008    90    40    57    92    15    73    11    86    80
    100    M      2009    21    98    48    31    84    73    85    94    87
    100    F      2007    15    44    95    75    90    73    58    16    54
    100    F      2008    93    25    50    68    8    25    36    91    78
    100    F      2009    95    36    3    59    67    57    34    59    75
```

Metadata can be added to an array at any time using:

```
[6]: arr.meta.description = 'array containing random values between 0 and 100'

arr.meta

[6]: title: random array
      description: array containing random values between 0 and 100
```

Warning:

Currently, only the HDF (.h5) file format supports saving and loading array metadata.

Metadata is not kept when actions or methods are applied on an array except for operations modifying the object in-place, such as `pop[age < 10] = 0`, and when the method `copy()` is called. Do not add metadata to an array if you know you will apply actions or methods on it before dumping it.

Array creation functions

Arrays can also be generated in an easier way through *creation functions*:

- `ndtest` : creates a test array with increasing numbers as data
- `empty` : creates an array but leaves its allocated memory unchanged (i.e., it contains “garbage”. Be careful !)
- `zeros`: fills an array with 0
- `ones` : fills an array with 1
- `full` : fills an array with a given value
- `sequence` : creates an array from an axis by iteratively applying a function to a given initial value.

Except for `ndtest`, a list of axes must be provided. Axes can be passed in different ways:

- as Axis objects
- as integers defining the lengths of auto-generated wildcard axes
- as a string : ‘sex=M,F;time=2007,2008,2009’ (name is optional)

- as pairs (name, labels)

Optionally, the type of data stored by the array can be specified using argument `dtype`.

```
[7]: # start defines the starting value of data
ndtest(['age=0..2', 'sex=M,F', 'time=2007..2009'], start=-1)
```

```
[7]: age  sex\time  2007  2008  2009
      0      M    -1     0     1
      0      F     2     3     4
      1      M     5     6     7
      1      F     8     9    10
      2      M    11    12    13
      2      F    14    15    16
```

```
[8]: # start defines the starting value of data
# label_start defines the starting index of labels
ndtest((3, 3), start=-1, label_start=2)
```

```
[8]: a\b  b2  b3  b4
      a2 -1  0  1
      a3  2  3  4
      a4  5  6  7
```

```
[9]: # empty generates uninitialised array with correct axes
# (much faster but use with care!).
# This not really random either, it just reuses a portion
# of memory that is available, with whatever content is there.
# Use it only if performance matters and make sure all data
# will be overridden.
empty(['age=0..2', 'sex=M,F', 'time=2007..2009'])
```

```
[9]: age  sex\time  ...
      0      M    ...
      0      F    ...
      1      M    ...
      1      F    ...
      2      M    ...
      2      F    ...
```

```
[10]: # example with anonymous axes
zeros(['0..2', 'M,F', '2007..2009'])
```

```
[10]: {0}  {1}\{2}  2007  2008  2009
      0      M    0.0   0.0   0.0
      0      F    0.0   0.0   0.0
      1      M    0.0   0.0   0.0
      1      F    0.0   0.0   0.0
      2      M    0.0   0.0   0.0
      2      F    0.0   0.0   0.0
```

```
[11]: # dtype=int forces to store int data instead of default float
ones(['age=0..2', 'sex=M,F', 'time=2007..2009'], dtype=int)
```

```
[11]: age  sex\time  2007  2008  2009
      0      M     1     1     1
      0      F     1     1     1
      1      M     1     1     1
      1      F     1     1     1
```

(continues on next page)

(continued from previous page)

2	M	1	1	1
2	F	1	1	1

```
[12]: full(['age=0..2', 'sex=M,F', 'time=2007..2009'], 1.23)
```

```
[12]: age  sex\time  2007  2008  2009
      0      M  1.23  1.23  1.23
      0      F  1.23  1.23  1.23
      1      M  1.23  1.23  1.23
      1      F  1.23  1.23  1.23
      2      M  1.23  1.23  1.23
      2      F  1.23  1.23  1.23
```

All the above functions exist in `*(func)_like*` variants which take axes from another array

```
[13]: ones_like(arr)
```

```
[13]: age  sex  time\other  A01  A02  A03  B01  B02  B03  C01  C02  C03
      0   M      2007      1    1    1    1    1    1    1    1    1
      0   M      2008      1    1    1    1    1    1    1    1    1
      0   M      2009      1    1    1    1    1    1    1    1    1
      0   F      2007      1    1    1    1    1    1    1    1    1
      0   F      2008      1    1    1    1    1    1    1    1    1
      ... ..      ...    ...    ...    ...    ...    ...    ...    ...
     100   M      2008      1    1    1    1    1    1    1    1    1
     100   M      2009      1    1    1    1    1    1    1    1    1
     100   F      2007      1    1    1    1    1    1    1    1    1
     100   F      2008      1    1    1    1    1    1    1    1    1
     100   F      2009      1    1    1    1    1    1    1    1    1
```

Create an array using the special `sequence` function (see link to documentation of `sequence` in API reference for more examples):

```
[14]: # With initial=1.0 and inc=0.5, we generate the sequence 1.0, 1.5, 2.0, 2.5, 3.0, ...
      sequence('sex=M,F', initial=1.0, inc=0.5)
```

```
[14]: sex      M      F
      1.0    1.5
```

Inspecting Array objects

```
[15]: # create a test array
      arr = ndtest([age, sex, time, other])
```

Get array summary : metadata + dimensions + description of axes + dtype + size in memory

```
[16]: arr.info
```

```
[16]: 101 x 2 x 3 x 9
      age [101]: 0 1 2 ... 98 99 100
      sex [2]: 'M' 'F'
      time [3]: 2007 2008 2009
      other [9]: 'A01' 'A02' 'A03' ... 'C01' 'C02' 'C03'
      dtype: int64
      memory used: 42.61 Kb
```


Get axes

```
[17]: arr.axes
[17]: AxisCollection([
    Axis([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
↪21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
↪42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
↪63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
↪84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100], 'age'),
    Axis(['M', 'F'], 'sex'),
    Axis([2007, 2008, 2009], 'time'),
    Axis(['A01', 'A02', 'A03', 'B01', 'B02', 'B03', 'C01', 'C02', 'C03'], 'other')
])
```

Get number of dimensions

```
[18]: arr.ndim
[18]: 4
```

Get length of each dimension

```
[19]: arr.shape
[19]: (101, 2, 3, 9)
```

Get total number of elements of the array

```
[20]: arr.size
[20]: 5454
```

Get type of internal data (int, float, ...)

```
[21]: arr.dtype
[21]: dtype('int64')
```

Get size in memory

```
[22]: arr.memory_used
[22]: '42.61 Kb'
```

Display the array in the viewer (graphical user interface) in read-only mode. This will open a new window and block execution of the rest of code until the windows is closed! Required PyQt installed.

```
view(arr)
```

Or load it in Excel:

```
arr.to_excel()
```

More on Array objects

To know how to save and load arrays in CSV, Excel or HDF format, please refer to the *Loading and Dumping Arrays* section of the tutorial.

See the *Array* section of the API Reference to explore all methods of Array objects.

Session

A `Session` object is a dictionary-like object used to gather several arrays, axes and groups. A session is particularly adapted to gather all input objects of a model or to gather the output arrays from different scenarios. Like with arrays, it is possible to associate metadata to sessions.

Creating Sessions

To create a session, you can first create an empty session and then populate it with arrays, axes and groups:

```
[23]: # create an empty session
s_pop = Session()

# add axes to the session
gender = Axis("gender=Male,Female")
s_pop.gender = gender
time = Axis("time=2013..2017")
s_pop.time = time

# add arrays to the session
s_pop.pop = zeros((gender, time))
s_pop.births = zeros((gender, time))
s_pop.deaths = zeros((gender, time))

# add metadata after creation
s_pop.meta.title = 'Demographic Model of Belgium'
s_pop.meta.description = 'Models the demography of Belgium'

# print content of the session
print(s_pop.summary())
```

```
Metadata:
  title: Demographic Model of Belgium
  description: Models the demography of Belgium
gender: gender ['Male' 'Female'] (2)
time: time [2013 2014 2015 2016 2017] (5)
pop: gender, time (2 x 5) [float64]
births: gender, time (2 x 5) [float64]
deaths: gender, time (2 x 5) [float64]
```

or you can create and populate a session in one step:

```
[24]: gender = Axis("gender=Male,Female")
time = Axis("time=2013..2017")

# create and populate a new session in one step
# Python <= 3.5
s_pop = Session([('gender', gender), ('time', time), ('pop', zeros((gender, time))),
```

(continues on next page)

(continued from previous page)

```

        ('births', zeros((gender, time))), ('deaths', zeros((gender, time))),
        meta=[('title', 'Demographic Model of Belgium'), ('description',
↪ 'Modelize the demography of Belgium')])
# Python 3.6+
s_pop = Session(gender=gender, time=time, pop=zeros((gender, time)),
                births=zeros((gender, time)), deaths=zeros((gender, time)),
                meta=Metadata(title='Demographic Model of Belgium', description=
↪ 'Modelize the demography of Belgium'))

# print content of the session
print(s_pop.summary())

```

```

Metadata:
  title: Demographic Model of Belgium
  description: Modelize the demography of Belgium
gender: gender ['Male' 'Female'] (2)
time: time [2013 2014 2015 2016 2017] (5)
pop: gender, time (2 x 5) [float64]
births: gender, time (2 x 5) [float64]
deaths: gender, time (2 x 5) [float64]

```

Warning:

Contrary to array metadata, saving and loading session metadata is supported for all current session file formats: Excel, CSV and HDF (.h5).

Metadata is not kept when actions or methods are applied on a session except for operations modifying a session in-place, such as: `s.arr1 = 0`. Do not add metadata to a session if you know you will apply actions or methods on it before dumping it.

More on Session objects

To know how to save and load sessions in CSV, Excel or HDF format, please refer to the [Loading and Dumping Sessions](#) section of the tutorial.

To see how to work with sessions, please read the [Working With Sessions](#) section of the tutorial.

Finally, see the [Session](#) section of the API Reference to explore all methods of Session objects.

4.2.3 Load And Dump Arrays, Sessions, Axes And Groups

The LArray library provides methods and functions to load and dump Array, Session, Axis Group objects to several formats such as Excel, CSV and HDF5. The HDF5 file format is designed to store and organize large amounts of data. It allows to read and write data much faster than when working with CSV and Excel files.

```

[1]: # first of all, import the LArray library
from larray import *

```

Loading and Dumping Arrays

Loading Arrays - Basic Usage (CSV, Excel, HDF5)

To read an array from a CSV file, you must use the `read_csv` function:

```
[2]: csv_dir = get_example_filepath('examples')

# read the array pop from the file 'pop.csv'.
# The data of the array below is derived from a subset of the demo_pjan table from
↳Eurostat
pop = read_csv(csv_dir + '/pop.csv')
pop

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-2-0ebfcb6cb1c0> in <module>
      3 # read the array pop from the file 'pop.csv'.
      4 # The data of the array below is derived from a subset of the demo_pjan table
↳from Eurostat
----> 5 pop = read_csv(csv_dir + '/pop.csv')
      6 pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
      700         else:
      701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
      703     return wrapper
      704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳**kwargs)
      231         raw = False
      232
--> 233     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳fill_value=fill_value, raw=raw, wide=wide)
      234
      235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳columns, raw, parse_header, wide, cartesian_prod, **kwargs)
      338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳' in axes_names[-1]
      339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳cartesian_prod=cartesian_prod, **kwargs)
      341
      342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳**kwargs)
```

(continues on next page)

(continued from previous page)

```

236     if cartesian_prod:
237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238                                     fill_value=fill_value,
↳ **kwargs)
239     else:
240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
    54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
    55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
    57     if isinstance(idx, pd.core.index.MultiIndex):
    58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

To read an array from a sheet of an Excel file, you can use the `read_excel` function:

```

[3]: filepath_excel = get_example_filepath('examples.xlsx')

# read the array from the sheet 'births' of the Excel file 'examples.xlsx'
# The data of the array below is derived from a subset of the demo_fasec table from
↳ Eurostat
↳ births = read_excel(filepath_excel, 'births')
births

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-3-03e491b54429> in <module>
      3 # read the array from the sheet 'births' of the Excel file 'examples.xlsx'
      4 # The data of the array below is derived from a subset of the demo_fasec_
↳ table from Eurostat
--> 5 births = read_excel(filepath_excel, 'births')
      6 births

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700     else:
    701         kwargs[new_arg_name] = new_arg_value
--> 702     return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)

```

(continues on next page)

(continued from previous page)

```

700         else:
701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
703     return wrapper
704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/excel.py in read_excel(filepath, sheet, nb_axes,
↳ index_col, fill_value, na, sort_rows, sort_columns, wide, engine, range, **kwargs)
222         df = pd.read_excel(filepath, sheet, index_col=index_col,
↳ engine=engine, **kwargs)
223         return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ raw=index_col is None,
--> 224             fill_value=fill_value, wide=wide)
225
226

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
236     if cartesian_prod:
237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238             fill_value=fill_value,
↳ **kwargs)
239     else:
240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
57     if isinstance(idx, pd.core.index.MultiIndex):
58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):

```

(continues on next page)

(continued from previous page)

```

44         if sort:
45             return list(idx.levels)

```

```
AttributeError: module 'pandas.core' has no attribute 'index'
```

The `open_excel` function in combination with the `load` method allows you to load several arrays from the same Workbook without opening and closing it several times:

```

# open the Excel file 'population.xlsx' and let it opened as long as you keep the
↪indent.
# The Python keyword ``with`` ensures that the Excel file is properly closed even if
↪an error occurs
with open_excel(filepath_excel) as wb:
    # load the array 'pop' from the sheet 'pop'
    pop = wb['pop'].load()
    # load the array 'births' from the sheet 'births'
    births = wb['births'].load()
    # load the array 'deaths' from the sheet 'deaths'
    deaths = wb['deaths'].load()

# the Workbook is automatically closed when getting out the block defined by the with
↪statement

```

Warning: `open_excel` requires to work on Windows and to have the library `xlwings` installed.

The HDF5 file format is specifically designed to store and organize large amounts of data. Reading and writing data in this file format is much faster than with CSV or Excel. An HDF5 file can contain multiple arrays, each array being associated with a key. To read an array from an HDF5 file, you must use the `read_hdf` function and provide the key associated with the array:

```

[4]: filepath_hdf = get_example_filepath('examples.h5')

# read the array from the file 'examples.h5' associated with the key 'deaths'
# The data of the array below is derived from a subset of the demo_magec table from
↪Eurostat
deaths = read_hdf(filepath_hdf, 'deaths')
deaths

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-4-a781a0c3b5ad> in <module>
      3 # read the array from the file 'examples.h5' associated with the key 'deaths'
      4 # The data of the array below is derived from a subset of the demo_magec
↪table from Eurostat
----> 5 deaths = read_hdf(filepath_hdf, 'deaths')
      6 deaths

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↪larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↪value, na, sort_rows, sort_columns, name, **kwargs)
      81         cartesian_prod = writer != 'LArray'
      82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↪columns, fill_value=fill_value,
--> 83                                parse_header=False, cartesian_prod=cartesian_
↪prod)
      84         if _meta is not None:

```

(continues on next page)

(continued from previous page)

```

85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
242             'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41         Returns unique labels for each dimension.
42         """
--> 43         if isinstance(idx, pd.core.index.MultiIndex):
44             if sort:
45                 return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

Dumping Arrays - Basic Usage (CSV, Excel, HDF5)

To write an array in a CSV file, you must use the `to_csv` method:

```

[5]: # save the array pop in the file 'pop.csv'
pop.to_csv('pop.csv')

-----
NameError                                Traceback (most recent call last)
<ipython-input-5-73b7c4cf8e59> in <module>
      1 # save the array pop in the file 'pop.csv'
----> 2 pop.to_csv('pop.csv')

NameError: name 'pop' is not defined

```

To write an array to a sheet of an Excel file, you can use the `to_excel` method:


```
[6]: # save the array pop in the sheet 'pop' of the Excel file 'population.xlsx'
pop.to_excel('population.xlsx', 'pop')

-----
NameError                                Traceback (most recent call last)
<ipython-input-6-be0820cd6131> in <module>
      1 # save the array pop in the sheet 'pop' of the Excel file 'population.xlsx'
----> 2 pop.to_excel('population.xlsx', 'pop')

NameError: name 'pop' is not defined
```

Note that `to_excel` create a new Excel file if it does not exist yet. If the file already exists, a new sheet is added after the existing ones if that sheet does not already exists:

```
[7]: # add a new sheet 'births' to the file 'population.xlsx' and save the array births in_
    ↪ it
births.to_excel('population.xlsx', 'births')

-----
NameError                                Traceback (most recent call last)
<ipython-input-7-b1bcd740e9ec> in <module>
      1 # add a new sheet 'births' to the file 'population.xlsx' and save the array_
    ↪ births in it
----> 2 births.to_excel('population.xlsx', 'births')

NameError: name 'births' is not defined
```

To reset an Excel file, you simply need to set the `overwrite_file` argument as `True`:

```
[8]: # 1. reset the file 'population.xlsx' (all sheets are removed)
# 2. create a sheet 'pop' and save the array pop in it
pop.to_excel('population.xlsx', 'pop', overwrite_file=True)

-----
NameError                                Traceback (most recent call last)
<ipython-input-8-e2b202392c1c> in <module>
      1 # 1. reset the file 'population.xlsx' (all sheets are removed)
      2 # 2. create a sheet 'pop' and save the array pop in it
----> 3 pop.to_excel('population.xlsx', 'pop', overwrite_file=True)

NameError: name 'pop' is not defined
```

The `open_excel` function in combination with the `dump()` method allows you to open a Workbook and to export several arrays at once. If the Excel file doesn't exist, the `overwrite_file` argument must be set to `True`.

Warning: The `save` method must be called at the end of the block defined by the `with` statement to actually write data in the Excel file, otherwise you will end up with an empty file.

```
# to create a new Excel file, argument overwrite_file must be set to True
with open_excel('population.xlsx', overwrite_file=True) as wb:
    # add a new sheet 'pop' and dump the array pop in it
    wb['pop'] = pop.dump()
    # add a new sheet 'births' and dump the array births in it
    wb['births'] = births.dump()
    # add a new sheet 'deaths' and dump the array deaths in it
    wb['deaths'] = deaths.dump()
```

(continues on next page)

(continued from previous page)

```
# actually write data in the Workbook
wb.save()

# the Workbook is automatically closed when getting out the block defined by the with_
↳statement
```

To write an array in an HDF5 file, you must use the `to_hdf` function and provide the key that will be associated with the array:

```
[9]: # save the array pop in the file 'population.h5' and associate it with the key 'pop'
pop.to_hdf('population.h5', 'pop')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-9-5e7420cb5ab4> in <module>
      1 # save the array pop in the file 'population.h5' and associate it with the_
↳key 'pop'
----> 2 pop.to_hdf('population.h5', 'pop')

NameError: name 'pop' is not defined
```

Specifying Wide VS Narrow format (CSV, Excel)

By default, all reading functions assume that arrays are stored in the wide format, meaning that their last axis is represented horizontally:

country \ time	2013	2014	2015
Belgium	11137974	11180840	11237274
France	65600350	65942267	66456279

By setting the `wide` argument to `False`, reading functions will assume instead that arrays are stored in the narrow format, i.e. one column per axis plus one value column:

country	time	value
Belgium	2013	11137974
Belgium	2014	11180840
Belgium	2015	11237274
France	2013	65600350
France	2014	65942267
France	2015	66456279

```
[10]: # set 'wide' argument to False to indicate that the array is stored in the 'narrow'
↳format
pop_BE_FR = read_csv(csv_dir + '/pop_narrow_format.csv', wide=False)
pop_BE_FR
```

```
-----
AttributeError                            Traceback (most recent call last)
<ipython-input-10-f355dbc17a36> in <module>
      1 # set 'wide' argument to False to indicate that the array is stored in the
↳'narrow' format
----> 2 pop_BE_FR = read_csv(csv_dir + '/pop_narrow_format.csv', wide=False)
```

(continues on next page)

(continued from previous page)

```

3 pop_BE_FR

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
    231         raw = False
    232
--> 233     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)
    234
    235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    316         series = df[df.columns[-1]]
    317         series.name = df.index.name
--> 318     return from_series(series, sort_rows=sort_columns, **kwargs)
    319
    320     # handle 1D arrays

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_series(s, sort_rows, fill_
↳ value, meta, **kwargs)
    120         a1    b1    6.0    7.0
    121         """
--> 122     if isinstance(s.index, pd.core.index.MultiIndex):
    123         # TODO: use argument sort=False when it will be available
    124         # (see https://github.com/pandas-dev/pandas/issues/15105)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```

[11]: # same for the read_excel function
pop_BE_FR = read_excel(filepath_excel, sheet='pop_narrow_format', wide=False)
pop_BE_FR

```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-11-c57b6e5ab300> in <module>
      1 # same for the read_excel function
----> 2 pop_BE_FR = read_excel(filepath_excel, sheet='pop_narrow_format', wide=False)
      3 pop_BE_FR

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702     return func(*args, **kwargs)

```

(continues on next page)

(continued from previous page)

```

703         return wrapper
704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
700         else:
701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
703     return wrapper
704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/excel.py in read_excel(filepath, sheet, nb_axes,
↳ index_col, fill_value, na, sort_rows, sort_columns, wide, engine, range, **kwargs)
222     df = pd.read_excel(filepath, sheet, index_col=index_col,
↳ engine=engine, **kwargs)
223     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ raw=index_col is None,
--> 224                             fill_value=fill_value, wide=wide)
225
226

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
316         series = df[df.columns[-1]]
317         series.name = df.index.name
--> 318         return from_series(series, sort_rows=sort_columns, **kwargs)
319
320     # handle 1D arrays

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_series(s, sort_rows, fill_
↳ value, meta, **kwargs)
120     a1    b1    6.0    7.0
121     """
--> 122     if isinstance(s.index, pd.core.index.MultiIndex):
123         # TODO: use argument sort=False when it will be available
124         # (see https://github.com/pandas-dev/pandas/issues/15105)

AttributeError: module 'pandas.core' has no attribute 'index'

```

By default, writing functions will set the name of the column containing the data to 'value'. You can choose the name of this column by using the `value_name` argument. For example, using `value_name='population'` you can export the previous array as:

country	time	population
Belgium	2013	11137974
Belgium	2014	11180840
Belgium	2015	11237274
France	2013	65600350
France	2014	65942267
France	2015	66456279

```
[12]: # dump the array pop_BE_FR in a narrow format (one column per axis plus one value_
      ↪column).
      # By default, the name of the column containing data is set to 'value'
      pop_BE_FR.to_csv('pop_narrow_format.csv', wide=False)

      # same but replace 'value' by 'population'
      pop_BE_FR.to_csv('pop_narrow_format.csv', wide=False, value_name='population')

-----
NameError                                Traceback (most recent call last)
<ipython-input-12-53c5a64c2ad7> in <module>
      1 # dump the array pop_BE_FR in a narrow format (one column per axis plus one_
      ↪value column).
      2 # By default, the name of the column containing data is set to 'value'
----> 3 pop_BE_FR.to_csv('pop_narrow_format.csv', wide=False)
      4
      5 # same but replace 'value' by 'population'

NameError: name 'pop_BE_FR' is not defined
```

```
[13]: # same for the to_excel method
      pop_BE_FR.to_excel('population.xlsx', 'pop_narrow_format', wide=False, value_name=
      ↪'population')

-----
NameError                                Traceback (most recent call last)
<ipython-input-13-9f8405f0a952> in <module>
      1 # same for the to_excel method
----> 2 pop_BE_FR.to_excel('population.xlsx', 'pop_narrow_format', wide=False, value_
      ↪name='population')

NameError: name 'pop_BE_FR' is not defined
```

Like with the `to_excel` method, it is possible to export arrays in a narrow format using `open_excel`. To do so, you must set the `wide` argument of the `dump` method to `False`:

```
with open_excel('population.xlsx') as wb:
    # dump the array pop_BE_FR in a narrow format:
    # one column per axis plus one value column.
    # Argument value_name can be used to change the name of the
    # column containing the data (default name is 'value')
    wb['pop_narrow_format'] = pop_BE_FR.dump(wide=False, value_name='population')
    # don't forget to call save()
    wb.save()

# in the sheet 'pop_narrow_format', data is written as:
# | country | time | value |
# | ----- | ---- | ----- |
# | Belgium | 2013 | 11137974 |
# | Belgium | 2014 | 11180840 |
# | Belgium | 2015 | 11237274 |
# | France  | 2013 | 65600350 |
# | France  | 2014 | 65942267 |
# | France  | 2015 | 66456279 |
```

Specifying Position in Sheet (Excel)

If you want to read an array from an Excel sheet which does not start at cell A1 (when there is more than one array stored in the same sheet for example), you will need to use the range argument.

Warning: Note that the range argument is only available if you have the library xlwings installed (Windows).

```
# the 'range' argument must be used to load data not starting at cell A1.
# This is useful when there is several arrays stored in the same sheet
births = read_excel(filepath_excel, sheet='pop_births_deaths', range='A9:E15')
```

Using open_excel, ranges are passed in brackets:

```
with open_excel(filepath_excel) as wb:
    # store sheet 'pop_births_deaths' in a temporary variable sh
    sh = wb['pop_births_deaths']
    # load the array pop from range A1:E7
    pop = sh['A1:E7'].load()
    # load the array births from range A9:E15
    births = sh['A9:E15'].load()
    # load the array deaths from range A17:E23
    deaths = sh['A17:E23'].load()

# the Workbook is automatically closed when getting out the block defined by the with_
↪statement
```

When exporting arrays to Excel files, data is written starting at cell A1 by default. Using the position argument of the to_excel method, it is possible to specify the top left cell of the dumped data. This can be useful when you want to export several arrays in the same sheet for example

Warning: Note that the position argument is only available if you have the library xlwings installed (Windows).

```
filename = 'population.xlsx'
sheetname = 'pop_births_deaths'

# save the arrays pop, births and deaths in the same sheet 'pop_births_and_deaths'.
# The 'position' argument is used to shift the location of the second and third_
↪arrays to be dumped
pop.to_excel(filename, sheetname)
births.to_excel(filename, sheetname, position='A9')
deaths.to_excel(filename, sheetname, position='A17')
```

Using open_excel, the position is passed in brackets (this allows you to also add extra informations):

```
with open_excel('population.xlsx') as wb:
    # add a new sheet 'pop_births_deaths' and write 'population' in the first cell
    # note: you can use wb['new_sheet_name'] = '' to create an empty sheet
    wb['pop_births_deaths'] = 'population'
    # store sheet 'pop_births_deaths' in a temporary variable sh
    sh = wb['pop_births_deaths']
    # dump the array pop in sheet 'pop_births_deaths' starting at cell A2
    sh['A2'] = pop.dump()
```

(continues on next page)

(continued from previous page)

```
# add 'births' in cell A10
sh['A10'] = 'births'
# dump the array births in sheet 'pop_births_deaths' starting at cell A11
sh['A11'] = births.dump()
# add 'deaths' in cell A19
sh['A19'] = 'deaths'
# dump the array deaths in sheet 'pop_births_deaths' starting at cell A20
sh['A20'] = deaths.dump()
# don't forget to call save()
wb.save()

# the Workbook is automatically closed when getting out the block defined by the with_
↪ statement
```

Exporting data without headers (Excel)

For some reasons, you may want to export only the data of an array without axes. For example, you may want to insert a new column containing extra information. As an exercise, let us consider we want to add the capital city for each country present in the array containing the total population by country:

country	capital city	2013	2014	2015
Belgium	Brussels	11137974	11180840	11237274
France	Paris	65600350	65942267	66456279
Germany	Berlin	80523746	80767463	81197537

Assuming you have prepared an excel sheet as below:

country	capital city	2013	2014	2015
Belgium	Brussels			
France	Paris			
Germany	Berlin			

you can then dump the data at right place by setting the header argument of `to_excel` to `False` and specifying the position of the data in sheet:

```
pop_by_country = pop.sum('gender')

# export only the data of the array pop_by_country starting at cell C2
pop_by_country.to_excel('population.xlsx', 'pop_by_country', header=False, position=
↪ 'C2')
```

Using `open_excel`, you can easily prepare the sheet and then export only data at the right place by either setting the header argument of the dump method to `False` or avoiding to call dump:

```
with open_excel('population.xlsx') as wb:
    # create new empty sheet 'pop_by_country'
    wb['pop_by_country'] = ''
    # store sheet 'pop_by_country' in a temporary variable sh
    sh = wb['pop_by_country']
    # write extra information (description)
    sh['A1'] = 'Population at 1st January by country'
    # export column names
```

(continues on next page)

(continued from previous page)

```

sh['A2'] = ['country', 'capital city']
sh['C2'] = pop_by_country.time.labels
# export countries as first column
sh['A3'].options(transpose=True).value = pop_by_country.country.labels
# export capital cities as second column
sh['B3'].options(transpose=True).value = ['Brussels', 'Paris', 'Berlin']
# export only data of pop_by_country
sh['C3'] = pop_by_country.dump(header=False)
# or equivalently
sh['C3'] = pop_by_country
# don't forget to call save()
wb.save()

# the Workbook is automatically closed when getting out the block defined by the with_
→ statement

```

Specifying the Number of Axes at Reading (CSV, Excel)

By default, `read_csv` and `read_excel` will search the position of the first cell containing the special character `\` in the header line in order to determine the number of axes of the array to read. The special character `\` is used to separate the name of the two last axes. If there is no special character `\`, `read_csv` and `read_excel` will consider that the array to read has only one dimension. For an array stored as:

country	gender \ time	2013	2014	2015
Belgium	Male	5472856	5493792	5524068
Belgium	Female	5665118	5687048	5713206
France	Male	31772665	31936596	32175328
France	Female	33827685	34005671	34280951
Germany	Male	39380976	39556923	39835457
Germany	Female	41142770	41210540	41362080

`read_csv` and `read_excel` will find the special character `\` in the second cell meaning it expects three axes (country, gender and time).

Sometimes, you need to read an array for which the name of the last axis is implicit:

country	gender	2013	2014	2015
Belgium	Male	5472856	5493792	5524068
Belgium	Female	5665118	5687048	5713206
France	Male	31772665	31936596	32175328
France	Female	33827685	34005671	34280951
Germany	Male	39380976	39556923	39835457
Germany	Female	41142770	41210540	41362080

For such case, you will have to inform `read_csv` and `read_excel` of the number of axes of the output array by setting the `nb_axes` argument:

```

[14]: # read the 3 x 2 x 3 array stored in the file 'pop_missing_axis_name.csv' without_
→ using 'nb_axes' argument.
pop = read_csv(csv_dir + '/pop_missing_axis_name.csv')
# shape and data type of the output array are not what we expected
pop.info

```



```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-14-46a6fc76fcd2> in <module>
      1 # read the 3 x 2 x 3 array stored in the file 'pop_missing_axis_name.csv'
      ↳ without using 'nb_axes' argument.
----> 2 pop = read_csv(csv_dir + '/pop_missing_axis_name.csv')
      3 # shape and data type of the output array are not what we expected
      4 pop.info

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
    231         raw = False
    232
--> 233     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)
    234
    235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338     unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339     res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                        unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    236     if cartesian_prod:
    237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238                        fill_value=fill_value,
↳ **kwargs)
    239     else:
    240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
    54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):

```

(continues on next page)

(continued from previous page)

```

55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
57     if isinstance(idx, pd.core.index.MultiIndex):
58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```

[15]: # by setting the 'nb_axes' argument, you can indicate to read_csv the number of axes,
↳ of the output array
pop = read_csv(csv_dir + '/pop_missing_axis_name.csv', nb_axes=3)

# give a name to the last axis
pop = pop.rename(-1, 'time')

# shape and data type of the output array are what we expected
pop.info

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-15-5e4ef9813b05> in <module>
      1 # by setting the 'nb_axes' argument, you can indicate to read_csv the number,
↳ of axes of the output array
--> 2 pop = read_csv(csv_dir + '/pop_missing_axis_name.csv', nb_axes=3)
      3
      4 # give a name to the last axis
      5 pop = pop.rename(-1, 'time')

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
700         else:
701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
703     return wrapper
704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
231         raw = False
232
--> 233     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)
234
235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)

```

(continues on next page)

(continued from previous page)

```

338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
236         if cartesian_prod:
237             df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238                 fill_value=fill_value,
↳ **kwargs)
239         else:
240             if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
57     if isinstance(idx, pd.core.index.MultiIndex):
58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```

[16]: # same for the read_excel function
pop = read_excel(filepath_excel, sheet='pop_missing_axis_name', nb_axes=3)
pop = pop.rename(-1, 'time')
pop.info

```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-16-9e97e7b5f94e> in <module>
      1 # same for the read_excel function
--> 2 pop = read_excel(filepath_excel, sheet='pop_missing_axis_name', nb_axes=3)
      3 pop = pop.rename(-1, 'time')
      4 pop.info

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)

```

(continues on next page)

(continued from previous page)

```

700         else:
701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
703     return wrapper
704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
700         else:
701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
703     return wrapper
704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/excel.py in read_excel(filepath, sheet, nb_axes,
↳ index_col, fill_value, na, sort_rows, sort_columns, wide, engine, range, **kwargs)
222     df = pd.read_excel(filepath, sheet, index_col=index_col,
↳ engine=engine, **kwargs)
223     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ raw=index_col is None,
--> 224                             fill_value=fill_value, wide=wide)
225
226

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338     unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
339     res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
236     if cartesian_prod:
237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238                             fill_value=fill_value,
↳ **kwargs)
239     else:
240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)

```

(continues on next page)

(continued from previous page)

```

57     if isinstance(idx, pd.core.index.MultiIndex):
58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

NaNs and Missing Data Handling at Reading (CSV, Excel)

Sometimes, there is no data available for some label combinations. In the example below, the rows corresponding to France - Male and Germany - Female are missing:

country	gender \ time	2013	2014	2015
Belgium	Male	5472856	5493792	5524068
Belgium	Female	5665118	5687048	5713206
France	Female	33827685	34005671	34280951
Germany	Male	39380976	39556923	39835457

By default, `read_csv` and `read_excel` will fill cells associated with missing label combinations with nans. Be aware that, in that case, an int array will be converted to a float array.

```

[17]: # by default, cells associated with missing label combinations are filled with nans.
      # In that case, the output array is converted to a float array
      read_csv(csv_dir + '/pop_missing_values.csv')

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-17-7598edd9f359> in <module>
      1 # by default, cells associated with missing label combinations are filled_
↳ with nans.
      2 # In that case, the output array is converted to a float array
--> 3 read_csv(csv_dir + '/pop_missing_values.csv')

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
700         else:
701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
703         return wrapper
704         return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
231         raw = False
232
--> 233         return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)

```

(continues on next page)

(continued from previous page)

```

234
235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    236     if cartesian_prod:
    237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238             fill_value=fill_value,
↳ **kwargs)
    239     else:
    240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
    54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
    55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
    57     if isinstance(idx, pd.core.index.MultiIndex):
    58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

However, it is possible to choose which value to use to fill missing cells using the `fill_value` argument:

```
[18]: read_csv(csv_dir + '/pop_missing_values.csv', fill_value=0)
```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-18-07a4dc5d864a> in <module>
----> 1 read_csv(csv_dir + '/pop_missing_values.csv', fill_value=0)

```

(continues on next page)

(continued from previous page)

```

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
    231     raw = False
    232
--> 233     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)
    234
    235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338     unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339     res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                        unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    236     if cartesian_prod:
    237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238                        fill_value=fill_value,
↳ **kwargs)
    239     else:
    240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
    54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
    55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
    57     if isinstance(idx, pd.core.index.MultiIndex):
    58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.

```

(continues on next page)

(continued from previous page)

```

42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```
[19]: # same for the read_excel function
```

```
read_excel(filepath_excel, sheet='pop_missing_values', fill_value=0)
```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-19-debca9e01abc> in <module>
      1 # same for the read_excel function
--> 2 read_excel(filepath_excel, sheet='pop_missing_values', fill_value=0)

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/excel.py in read_excel(filepath, sheet, nb_axes,
↳ index_col, fill_value, na, sort_rows, sort_columns, wide, engine, range, **kwargs)
    222     df = pd.read_excel(filepath, sheet, index_col=index_col,
↳ engine=engine, **kwargs)
    223     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ raw=index_col is None,
--> 224                             fill_value=fill_value, wide=wide)
    225
    226

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338     unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339     res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_row
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)

```

(continues on next page)

(continued from previous page)

```

236     if cartesian_prod:
237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238                                     fill_value=fill_value,
↳ **kwargs)
239     else:
240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
    54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
    55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
    57     if isinstance(idx, pd.core.index.MultiIndex):
    58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

Sorting Axes at Reading (CSV, Excel, HDF5)

The `sort_rows` and `sort_columns` arguments of the reading functions allows you to sort rows and columns alphabetically:

```
[20]: # sort labels at reading --> Male and Female labels are inverted
read_csv(csv_dir + '/pop.csv', sort_rows=True)
```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-20-2a097cec7595> in <module>
      1 # sort labels at reading --> Male and Female labels are inverted
--> 2 read_csv(csv_dir + '/pop.csv', sort_rows=True)

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700     else:
    701         kwargs[new_arg_name] = new_arg_value
--> 702     return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
    231     raw = False

```

(continues on next page)

(continued from previous page)

```

232
--> 233     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)
234
235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338     unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
339     res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340         unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
236     if cartesian_prod:
237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238         fill_value=fill_value,
↳ **kwargs)
239     else:
240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):
55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
57     if isinstance(idx, pd.core.index.MultiIndex):
58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45         return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```
[21]: read_excel(filepath_excel, sheet='births', sort_rows=True)
```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-21-10dae479fc29> in <module>
----> 1 read_excel(filepath_excel, sheet='births', sort_rows=True)

```

(continues on next page)

(continued from previous page)

```

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703     return wrapper
    704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/excel.py in read_excel(filepath, sheet, nb_axes,
↳ index_col, fill_value, na, sort_rows, sort_columns, wide, engine, range, **kwargs)
    222     df = pd.read_excel(filepath, sheet, index_col=index_col,
↳ engine=engine, **kwargs)
    223     return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ raw=index_col is None,
--> 224                             fill_value=fill_value, wide=wide)
    225
    226

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338     unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339     res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    236     if cartesian_prod:
    237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
↳ columns=sort_columns,
--> 238                             fill_value=fill_value,
↳ **kwargs)
    239     else:
    240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
↳ sort_columns, fill_value, **kwargs)
    54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
↳ value=nan, **kwargs):

```

(continues on next page)

(continued from previous page)

```

55     idx = df.index
--> 56     labels = index_to_labels(idx, sort=sort_rows)
57     if isinstance(idx, pd.core.index.MultiIndex):
58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```
[22]: read_hdf(filepath_hdf, key='deaths').sort_axes()
```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-22-378ea28c9b5a> in <module>
----> 1 read_hdf(filepath_hdf, key='deaths').sort_axes()

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
81         cartesian_prod = writer != 'LArray'
82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83         parse_header=False, cartesian_prod=cartesian_
↳ prod)
84         if _meta is not None:
85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340         unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
242         'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

```

(continues on next page)

(continued from previous page)

```
~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'
```

Metadata (HDF5)

Since the version 0.29 of LArray, it is possible to add metadata to arrays:

```
[23]: pop.meta.title = 'Population at 1st January'
pop.meta.origin = 'Table demo_jpan from Eurostat'

pop.info

-----
NameError                                Traceback (most recent call last)
<ipython-input-23-ab1445aa42ff> in <module>
----> 1 pop.meta.title = 'Population at 1st January'
      2 pop.meta.origin = 'Table demo_jpan from Eurostat'
      3
      4 pop.info

NameError: name 'pop' is not defined
```

These metadata are automatically saved and loaded when working with the HDF5 file format:

```
[24]: pop.to_hdf('population.h5', 'pop')

new_pop = read_hdf('population.h5', 'pop')
new_pop.info

-----
NameError                                Traceback (most recent call last)
<ipython-input-24-bafaf422e3d9> in <module>
----> 1 pop.to_hdf('population.h5', 'pop')
      2
      3 new_pop = read_hdf('population.h5', 'pop')
      4 new_pop.info

NameError: name 'pop' is not defined
```

Warning: Currently, metadata associated with arrays cannot be saved and loaded when working with CSV and Excel files. This restriction does not apply however to metadata associated with sessions.

Loading and Dumping Sessions

One of the main advantages of grouping arrays, axes and groups in session objects is that you can load and save all of them in one shot. Like arrays, it is possible to associate metadata to a session. These can be saved and loaded in all file formats.

Loading Sessions (CSV, Excel, HDF5)

To load the items of a session, you have two options:

- 1) Instantiate a new session and pass the path to the Excel/HDF5 file or to the directory containing CSV files to the Session constructor:

```
[25]: # create a new Session object and load all arrays, axes, groups and metadata
# from all CSV files located in the passed directory
csv_dir = get_example_filepath('demography_eurostat')
session = Session(csv_dir)

# create a new Session object and load all arrays, axes, groups and metadata
# stored in the passed Excel file
filepath_excel = get_example_filepath('demography_eurostat.xlsx')
session = Session(filepath_excel)

# create a new Session object and load all arrays, axes, groups and metadata
# stored in the passed HDF5 file
filepath_hdf = get_example_filepath('demography_eurostat.h5')
session = Session(filepath_hdf)

print(session.summary())
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-25-a92fe605d750> in <module>
      2 # from all CSV files located in the passed directory
      3 csv_dir = get_example_filepath('demography_eurostat')
----> 4 session = Session(csv_dir)
      5
      6 # create a new Session object and load all arrays, axes, groups and metadata

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
--> 96         self.load(a0)
      97     else:
      98         # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
     426     else:
     427         handler = handler_cls(fname)
--> 428     metadata, objects = handler.read(names, display=display, **kwargs)
     429     for k, v in objects.items():
     430         self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
(continues on next page)
```

(continued from previous page)

```

119         ignore_exceptions = kwargs.pop('ignore_exceptions', False)
120         self._open_for_read()
--> 121         metadata = self._read_metadata()
122         key_types = self.list_items()
123         if keys is not None:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in _read_metadata(self)
    327         filepath = self._to_filepath('__metadata__')
    328         if os.path.isfile(filepath):
--> 329             meta = read_csv(filepath, wide=False)
    330             return Metadata.from_array(meta)
    331         else:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
    700         else:
    701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
    703         return wrapper
    704         return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
    231         raw = False
    232
--> 233         return df_asarray(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ fill_value=fill_value, raw=raw, wide=wide)
    234
    235

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    316         series = df[df.columns[-1]]
    317         series.name = df.index.name
--> 318         return from_series(series, sort_rows=sort_columns, **kwargs)
    319
    320         # handle 1D arrays

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_series(s, sort_rows, fill_
↳ value, meta, **kwargs)
    120         a1    b1    6.0    7.0
    121         """
--> 122         if isinstance(s.index, pd.core.index.MultiIndex):
    123             # TODO: use argument sort=False when it will be available
    124             # (see https://github.com/pandas-dev/pandas/issues/15105)

AttributeError: module 'pandas.core' has no attribute 'index'

```

- 2) Call the load method on an existing session and pass the path to the Excel/HDF5 file or to the directory containing CSV files as first argument:

```
[26]: # create a session containing 3 axes, 2 groups and one array 'pop'
filepath = get_example_filepath('pop_only.xlsx')
session = Session(filepath)

print(session.summary())

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-26-56elec09flab> in <module>
      1 # create a session containing 3 axes, 2 groups and one array 'pop'
      2 filepath = get_example_filepath('pop_only.xlsx')
----> 3 session = Session(filepath)
      4
      5 print(session.summary())

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
    94         if isinstance(a0, str):
    95             # assume a0 is a filename
---> 96             self.load(a0)
    97         else:
    98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
display, **kwargs)
    426         else:
    427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
    429             for k, v in objects.items():
    430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
    128             print("loading", type, "object", key, "...", end=' ')
    129             try:
--> 130                 res[key] = self._read_item(key, type, *args, **kwargs)
    131             except Exception:
    132                 if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/inout/excel.py in _read_item(self, key, type, *args,
**kwargs)
    252         if type == 'Array':
    253             df = self.handle.parse(key, *args, **kwargs)
--> 254             return df_asarray(df, raw=True)
    255         else:
    256             raise TypeError()

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
cartesian_prod=cartesian_prod, **kwargs)
    341
```

(continues on next page)

(continued from previous page)

```

342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:0'
    ↪ 'x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
    ↪ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
    ↪ **kwargs)
    236     if cartesian_prod:
    237         df, axes_labels = cartesian_product_df(df, sort_rows=sort_rows, sort_
    ↪ columns=sort_columns,
--> 238                                     fill_value=fill_value,
    ↪ **kwargs)
    239     else:
    240         if sort_rows or sort_columns:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/inout/pandas.py in cartesian_product_df(df, sort_rows,
    ↪ sort_columns, fill_value, **kwargs)
    54 def cartesian_product_df(df, sort_rows=False, sort_columns=False, fill_
    ↪ value=nan, **kwargs):
    55     idx = df.index
    56     labels = index_to_labels(idx, sort=sort_rows)
    57     if isinstance(idx, pd.core.index.MultiIndex):
    58         if sort_rows:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```

[27]: # call the load method on the previous session and add the 'births' and 'deaths'
    ↪ arrays to it
    filepath = get_example_filepath('births_and_deaths.xlsx')
    session.load(filepath)

    print(session.summary())

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-27-fe4df71a7d14> in <module>
      1 # call the load method on the previous session and add the 'births' and
    ↪ 'deaths' arrays to it
      2 filepath = get_example_filepath('births_and_deaths.xlsx')
--> 3 session.load(filepath)
      4
      5 print(session.summary())

NameError: name 'session' is not defined

```

The load method offers some options:

- 1) Using the names argument, you can specify which items to load:

```
[28]: session = Session()

# use the names argument to only load births and deaths arrays
session.load(filepath_hdf, names=['births', 'deaths'])

print(session.summary())

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-28-8f78684c5696> in <module>
      2
      3 # use the names argument to only load births and deaths arrays
----> 4 session.load(filepath_hdf, names=['births', 'deaths'])
      5
      6 print(session.summary())

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine, _
↳ display, **kwargs)
      426         else:
      427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
      429             for k, v in objects.items():
      430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
      128             print("loading", type, "object", key, "...", end=' ')
      129             try:
--> 130                 res[key] = self._read_item(key, type, *args, **kwargs)
      131             except Exception:
      132                 if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args, _
↳ **kwargs)
      137         else:
      138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
      140
      141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
      81             cartesian_prod = writer != 'LArray'
      82             res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83             parse_header=False, cartesian_prod=cartesian_
↳ prod)
      84             if _meta is not None:
      85                 res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
      338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳ ' in axes_names[-1]
```

(continues on next page)

(continued from previous page)

```

339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
242             'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245     # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

2) Setting the display argument to True, the load method will print a message each time a new item is loaded:

```

[29]: session = Session()

# with display=True, the load method will print a message
# each time a new item is loaded
session.load(filepath_hdf, display=True)

opening /home/docs/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/
↳ python3.6/site-packages/larray-0.32-py3.6.egg/larray/tests/data/examples.h5
loading Array object births ...

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-29-be27048b30bd> in <module>
      3 # with display=True, the load method will print a message
      4 # each time a new item is loaded
----> 5 session.load(filepath_hdf, display=True)

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
426     else:
427         handler = handler_cls(fname)
--> 428         metadata, objects = handler.read(names, display=display, **kwargs)
429         for k, v in objects.items():
430             self[k] = v

```

(continues on next page)

(continued from previous page)

```

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
    128         print("loading", type, "object", key, "...", end=' ')
    129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
    131         except Exception:
    132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
    137         else:
    138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
    140
    141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
    81         cartesian_prod = writer != 'LArray'
    82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83                             parse_header=False, cartesian_prod=cartesian_
↳ prod)
    84         if _meta is not None:
    85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
    242                             'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
    244
    245         # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)

```

(continues on next page)

(continued from previous page)

```

41     Returns unique labels for each dimension.
42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

Dumping Sessions (CSV, Excel, HDF5)

To save a session, you need to call the save method. The first argument is the path to a Excel/HDF5 file or to a directory if items are saved to CSV files:

```

[30]: # save items of a session in CSV files.
# Here, the save method will create a 'population' directory in which CSV files will
↳be written
session.save('population')

# save session to an HDF5 file
session.save('population.h5')

# save session to an Excel file
session.save('population.xlsx')

# load session saved in 'population.h5' to see its content
Session('population.h5')

-----
IndexError                                Traceback (most recent call last)
<ipython-input-30-20aa34ff2da2> in <module>
      7
      8 # save session to an Excel file
--> 9 session.save('population.xlsx')
     10
     11 # load session saved in 'population.h5' to see its content

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in save(self, fname, names, engine,
↳overwrite, display, **kwargs)
     495         names_set = set(names)
     496         items = [(k, v) for k, v in items if k in names_set]
--> 497         handler.dump(meta, items, display=display, **kwargs)
     498
     499     def to_globals(self, names=None, depth=0, warn=True, inplace=False):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/common.py in dump(self, metadata, key_values,
↳*args, **kwargs)
     170         print("Cannot dump {}. {} is not a supported type".
↳format(key, type(value).__name__))
     171         self.save()
--> 172         self.close()
     173         self._update_original_file()

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/excel.py in close(self)

```

(continues on next page)

(continued from previous page)

```

280
281     def close(self):
--> 282         self.handle.close()
283
284

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ pandas/io/excel/_base.py in close(self)
    779     def close(self):
    780         """synonym for save, to make it more file-like"""
--> 781         return self.save()
    782
    783

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ pandas/io/excel/_openpyxl.py in save(self)
    41         Save workbook to disk.
    42         """
---> 43         return self.book.save(self.path)
    44
    45     @classmethod

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ openpyxl/workbook/workbook.py in save(self, filename)
    390         if self.write_only and not self.worksheets:
    391             self.create_sheet()
--> 392         save_workbook(self, filename)
    393
    394

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ openpyxl/writer/excel.py in save_workbook(workbook, filename)
    291         archive = ZipFile(filename, 'w', ZIP_DEFLATED, allowZip64=True)
    292         writer = ExcelWriter(workbook, archive)
--> 293         writer.save()
    294         return True
    295

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ openpyxl/writer/excel.py in save(self)
    273     def save(self):
    274         """Write data into the archive."""
--> 275         self.write_data()
    276         self._archive.close()
    277

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ openpyxl/writer/excel.py in write_data(self)
    87         writer = WorkbookWriter(self.workbook)
    88         archive.writestr(ARC_ROOT_RELS, writer.write_root_rels())
---> 89         archive.writestr(ARC_WORKBOOK, writer.write())
    90         archive.writestr(ARC_WORKBOOK_RELS, writer.write_rels())
    91

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ openpyxl/workbook/_writer.py in write(self)
    146         self.write_names()

```

(continues on next page)

(continued from previous page)

```

147         self.write_pivots()
--> 148         self.write_views()
149         self.write_refs()
150

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ openpyxl/workbook/_writer.py in write_views(self)
133
134     def write_views(self):
--> 135         active = get_active_sheet(self.wb)
136         if self.wb.views:
137             self.wb.views[0].activeTab = active

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ openpyxl/workbook/_writer.py in get_active_sheet(wb)
31     visible_sheets = [idx for idx, sheet in enumerate(wb._sheets) if sheet.
↳ sheet_state == "visible"]
32     if not visible_sheets:
---> 33         raise IndexError("At least one sheet must be visible")
34
35     idx = wb._active_sheet_index

IndexError: At least one sheet must be visible

```

Note: Concerning the CSV and Excel formats, the metadata is saved in one Excel sheet (CSV file) named `__metadata__ (.csv)`. This sheet (CSV file) name cannot be changed.

The save method has several arguments:

- 1) Using the `names` argument, you can specify which items to save:

```

[31]: # use the names argument to only save births and deaths arrays
session.save('population.h5', names=['births', 'deaths'])

# load session saved in 'population.h5' to see its content
Session('population.h5')

[31]: Session()

```

- 2) By default, dumping a session to an Excel or HDF5 file will overwrite it. By setting the `overwrite` argument to `False`, you can choose to update the existing Excel or HDF5 file:

```

[32]: pop = read_csv('./population/pop.csv')
ses_pop = Session([('pop', pop)])

# by setting overwrite to False, the destination file is updated instead of_
↳ overwritten.
# The items already stored in the file but not present in the session are left intact.
# On the contrary, the items that exist in both the file and the session are_
↳ completely overwritten.
ses_pop.save('population.h5', overwrite=False)

# load session saved in 'population.h5' to see its content
Session('population.h5')

```

```

-----
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-32-ae724498303f> in <module>
----> 1 pop = read_csv('./population/pop.csv')
      2 ses_pop = Session([('pop', pop)])
      3
      4 # by setting overwrite to False, the destination file is updated instead of_
↳ overwritten.
      5 # The items already stored in the file but not present in the session are_
↳ left intact.

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/util/misc.py in wrapper(*args, **kwargs)
      700         else:
      701             kwargs[new_arg_name] = new_arg_value
--> 702         return func(*args, **kwargs)
      703     return wrapper
      704     return _deprecate_kwarg

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/csv.py in read_csv(filepath_or_buffer, nb_axes,
↳ index_col, sep, headersep, fill_value, na, sort_rows, sort_columns, wide, dialect,
↳ **kwargs)
      214         index_col = [0]
      215
--> 216     df = pd.read_csv(filepath_or_buffer, index_col=index_col, sep=sep,
↳ **kwargs)
      217     if dialect == 'liam2':
      218         if len(df) == 1:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ pandas/io/parsers.py in parser_f(filepath_or_buffer, sep, delimiter, header, names,
↳ index_col, usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine, converters,
↳ true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows, na_values,
↳ keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_
↳ format, keep_date_col, date_parser, dayfirst, cache_dates, iterator, chunksize,
↳ compression, thousands, decimal, lineterminator, quotechar, quoting, doublequote,
↳ escapechar, comment, encoding, dialect, error_bad_lines, warn_bad_lines, delim
↳ whitespace, low_memory, memory_map, float_precision)
      674         )
      675
--> 676     return _read(filepath_or_buffer, kwds)
      677
      678     parser_f.__name__ = name

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ pandas/io/parsers.py in _read(filepath_or_buffer, kwds)
      446
      447     # Create the parser.
--> 448     parser = TextFileReader(fp_or_buf, **kwds)
      449
      450     if chunksize or iterator:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ pandas/io/parsers.py in __init__(self, f, engine, **kwds)
      878         self.options["has_index_names"] = kwds["has_index_names"]
      879

```

(continues on next page)

(continued from previous page)

```

--> 880         self._make_engine(self.engine)
      881
      882     def close(self):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳pandas/io/parsers.py in _make_engine(self, engine)
      1112     def _make_engine(self, engine="c"):
      1113         if engine == "c":
-> 1114             self._engine = CParserWrapper(self.f, **self.options)
      1115     else:
      1116         if engine == "python":

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳pandas/io/parsers.py in __init__(self, src, **kwds)
      1889         kwds["usecols"] = self.usecols
      1890
-> 1891         self._reader = parsers.TextReader(src, **kwds)
      1892         self.unnamed_cols = self._reader.unnamed_cols
      1893

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.__cinit__()

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._setup_parser_source()

FileNotFoundError: [Errno 2] File ./population/pop.csv does not exist: './population/
↳pop.csv'

```

3) Setting the display argument to True, the save method will print a message each time an item is dumped:

```

[33]: # with display=True, the save method will print a message
      # each time an item is dumped
      session.save('population.h5', display=True)

```

4.2.4 Transforming Arrays (Relabeling, Renaming, Reordering, Combining, Extending, Sorting, ...)

Import the LArray library:

```

[1]: from larray import *

```

```

[2]: # load the 'demography_eurostat' dataset
      demo_eurostat = load_example_data('demography_eurostat')

```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-2-43cbc3a4a95d> in <module>
      1 # load the 'demography_eurostat' dataset
----> 2 demo_eurostat = load_example_data('demography_eurostat')

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
↳format(list(AVAILABLE_EXAMPLE_DATA.keys())))
----> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

```

(continues on next page)

(continued from previous page)

```

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
    94         if isinstance(a0, str):
    95             # assume a0 is a filename
---> 96             self.load(a0)
    97         else:
    98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
    426         else:
    427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
    429             for k, v in objects.items():
    430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
    128             print("loading", type, "object", key, "...", end=' ')
    129             try:
--> 130                 res[key] = self._read_item(key, type, *args, **kwargs)
    131             except Exception:
    132                 if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
    137         else:
    138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
    140
    141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
    81             cartesian_prod = writer != 'LArray'
    82             res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83                             parse_header=False, cartesian_prod=cartesian_
↳ prod)
    84             if _meta is not None:
    85                 res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338             unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339             res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342             # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

```

(continues on next page)

(continued from previous page)

```
~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used_
↳ when cartesian_prod is set to False. '
    242         'Please call the method sort_axes on the_
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
    244
    245     # Pandas treats column labels as column names (strings) so we need to_
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'
```

```
[3]: # get a copy of the 'pop' array from the 'demography_eurostat' dataset
pop = demo_eurostat.pop.copy()
pop
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-05b1a30c03e6> in <module>
      1 # get a copy of the 'pop' array from the 'demography_eurostat' dataset
--> 2 pop = demo_eurostat.pop.copy()
      3 pop

NameError: name 'demo_eurostat' is not defined
```

Manipulating axes

The Array class offers several methods to manipulate the axes and labels of an array:

- *set_labels*: to replace all or some labels of one or several axes.
- *rename*: to replace one or several axis names.
- *set_axes*: to replace one or several axes.
- *transpose*: to modify the order of axes.
- *drop*: to remove one or several labels.
- *combine_axes*: to combine axes.
- *split_axes*: to split one or several axes by splitting their labels and names.
- *reindex*: to reorder, add and remove labels of one or several axes.
- *insert*: to insert a label at a given position.

Relabeling

Replace some labels of an axis:

```
[4]: # replace all labels of the 'gender' axis by passing the list of all new labels
pop_new_labels = pop.set_labels('gender', ['Men', 'Women'])
pop_new_labels
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-623309d162e8> in <module>
      1 # replace all labels of the 'gender' axis by passing the list of all new_
    ↪ labels
----> 2 pop_new_labels = pop.set_labels('gender', ['Men', 'Women'])
      3 pop_new_labels

NameError: name 'pop' is not defined
```

```
[5]: # set all labels of the 'country' axis to uppercase by passing the function str.
    ↪ upper()
pop_new_labels = pop.set_labels('country', str.upper)
pop_new_labels
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-5-2c6df1fd25e2> in <module>
      1 # set all labels of the 'country' axis to uppercase by passing the function_
    ↪ str.upper()
----> 2 pop_new_labels = pop.set_labels('country', str.upper)
      3 pop_new_labels

NameError: name 'pop' is not defined
```

```
[6]: # replace only one label of the 'gender' axis by passing a dict
pop_new_labels = pop.set_labels('gender', {'Male': 'Men'})
pop_new_labels
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-6-44062fe81d08> in <module>
      1 # replace only one label of the 'gender' axis by passing a dict
----> 2 pop_new_labels = pop.set_labels('gender', {'Male': 'Men'})
      3 pop_new_labels

NameError: name 'pop' is not defined
```

See [set_labels](#) for more details and examples.

Renaming axes

Rename one axis:

```
[7]: # 'rename' returns a copy of the array
pop_new_names = pop.rename('time', 'year')
pop_new_names

-----
NameError                                Traceback (most recent call last)
<ipython-input-7-ebe3a3298518> in <module>
      1 # 'rename' returns a copy of the array
----> 2 pop_new_names = pop.rename('time', 'year')
      3 pop_new_names

NameError: name 'pop' is not defined
```

Rename several axes at once:

```
[8]: pop_new_names = pop.rename({'gender': 'sex', 'time': 'year'})
pop_new_names

-----
NameError                                Traceback (most recent call last)
<ipython-input-8-1c5ebbf525cf> in <module>
----> 1 pop_new_names = pop.rename({'gender': 'sex', 'time': 'year'})
      2 pop_new_names

NameError: name 'pop' is not defined
```

See *rename* for more details and examples.

Replacing Axes

Replace one axis:

```
[9]: new_gender = Axis('sex=Men,Women')
pop_new_axis = pop.set_axes('gender', new_gender)
pop_new_axis

-----
NameError                                Traceback (most recent call last)
<ipython-input-9-9465ac881bab> in <module>
      1 new_gender = Axis('sex=Men,Women')
----> 2 pop_new_axis = pop.set_axes('gender', new_gender)
      3 pop_new_axis

NameError: name 'pop' is not defined
```

Replace several axes at once:

```
[10]: new_country = Axis('country_codes=BE,FR,DE')
pop_new_axes = pop.set_axes({'country': new_country, 'gender': new_gender})
pop_new_axes

-----
NameError                                Traceback (most recent call last)
<ipython-input-10-01acccdf346c> in <module>
```

(continues on next page)

(continued from previous page)

```

1 new_country = Axis('country_codes=BE,FR,DE')
----> 2 pop_new_axes = pop.set_axes({'country': new_country, 'gender': new_gender})
3 pop_new_axes

```

```
NameError: name 'pop' is not defined
```

Reordering axes

Axes can be reordered using transpose method. By default, *transpose* reverse axes, otherwise it permutes the axes according to the list given as argument. Axes not mentioned come after those which are mentioned (and keep their relative order). Finally, *transpose* returns a copy of the array.

```
[11]: # starting order : country, gender, time
pop
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-11-8ec305916275> in <module>
      1 # starting order : country, gender, time
----> 2 pop

NameError: name 'pop' is not defined

```

```
[12]: # no argument --> reverse all axes
pop_transposed = pop.transpose()

# .T is a shortcut for .transpose()
pop_transposed = pop.T

pop_transposed
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-12-7225533ef09b> in <module>
      1 # no argument --> reverse all axes
----> 2 pop_transposed = pop.transpose()
      3
      4 # .T is a shortcut for .transpose()
      5 pop_transposed = pop.T

NameError: name 'pop' is not defined

```

```
[13]: # reorder according to list
pop_transposed = pop.transpose('gender', 'country', 'time')
pop_transposed
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-13-691cd72c0f5b> in <module>
      1 # reorder according to list
----> 2 pop_transposed = pop.transpose('gender', 'country', 'time')
      3 pop_transposed

NameError: name 'pop' is not defined

```

```
[14]: # move 'time' axis at first place
# not mentioned axes come after those which are mentioned (and keep their relative_
      ↳order)
pop_transposed = pop.transpose('time')
pop_transposed
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-14-0a5906d260bc> in <module>
      1 # move 'time' axis at first place
      2 # not mentioned axes come after those which are mentioned (and keep their_
      ↳relative order)
----> 3 pop_transposed = pop.transpose('time')
      4 pop_transposed

NameError: name 'pop' is not defined
```

```
[15]: # move 'gender' axis at last place
# not mentioned axes come before those which are mentioned (and keep their relative_
      ↳order)
pop_transposed = pop.transpose(..., 'gender')
pop_transposed
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-15-a70bb6241c42> in <module>
      1 # move 'gender' axis at last place
      2 # not mentioned axes come before those which are mentioned (and keep their_
      ↳relative order)
----> 3 pop_transposed = pop.transpose(..., 'gender')
      4 pop_transposed

NameError: name 'pop' is not defined
```

See [transpose](#) for more details and examples.

Dropping Labels

```
[16]: pop_labels_dropped = pop.drop([2014, 2016])
pop_labels_dropped
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-16-1d801a07a183> in <module>
----> 1 pop_labels_dropped = pop.drop([2014, 2016])
      2 pop_labels_dropped

NameError: name 'pop' is not defined
```

See [drop](#) for more details and examples.

Combine And Split Axes

Combine two axes:

```
[17]: pop_combined_axes = pop.combine_axes(('country', 'gender'))
pop_combined_axes

-----
NameError                                Traceback (most recent call last)
<ipython-input-17-a8b2b6532b26> in <module>
----> 1 pop_combined_axes = pop.combine_axes(('country', 'gender'))
      2 pop_combined_axes

NameError: name 'pop' is not defined
```

Split an axis:

```
[18]: pop_split_axes = pop_combined_axes.split_axes('country_gender')
pop_split_axes

-----
NameError                                Traceback (most recent call last)
<ipython-input-18-448027490424> in <module>
----> 1 pop_split_axes = pop_combined_axes.split_axes('country_gender')
      2 pop_split_axes

NameError: name 'pop_combined_axes' is not defined
```

See *combine_axes* and *split_axes* for more details and examples.

Reordering, adding and removing labels

The *reindex* method allows to reorder, add and remove labels along one axis:

```
[19]: # reverse years + remove 2013 + add 2018 + copy data for 2017 to 2018
pop_new_time = pop.reindex('time', '2018..2014', fill_value=pop[2017])
pop_new_time

-----
NameError                                Traceback (most recent call last)
<ipython-input-19-861f69ddf561> in <module>
      1 # reverse years + remove 2013 + add 2018 + copy data for 2017 to 2018
----> 2 pop_new_time = pop.reindex('time', '2018..2014', fill_value=pop[2017])
      3 pop_new_time

NameError: name 'pop' is not defined
```

or several axes:

```
[20]: pop_new = pop.reindex({'country': 'country=Luxembourg,Belgium,France,Germany',
                             'time': 'time=2018..2014'}, fill_value=0)
pop_new

-----
NameError                                Traceback (most recent call last)
<ipython-input-20-d5de989749ed> in <module>
----> 1 pop_new = pop.reindex({'country': 'country=Luxembourg,Belgium,France,Germany',
      2                             'time': 'time=2018..2014'}, fill_value=0)
```

(continues on next page)

(continued from previous page)

```
3 pop_new
```

```
NameError: name 'pop' is not defined
```

See *reindex* for more details and examples.

Another way to insert new labels is to use the `insert` method:

```
[21]: # insert a new country before 'France' with all values set to 0
pop_new_country = pop.insert(0, before='France', label='Luxembourg')
# or equivalently
pop_new_country = pop.insert(0, after='Belgium', label='Luxembourg')

pop_new_country

-----
NameError                                Traceback (most recent call last)
<ipython-input-21-c1e4f8f51f62> in <module>
      1 # insert a new country before 'France' with all values set to 0
----> 2 pop_new_country = pop.insert(0, before='France', label='Luxembourg')
      3 # or equivalently
      4 pop_new_country = pop.insert(0, after='Belgium', label='Luxembourg')
      5

NameError: name 'pop' is not defined
```

See *insert* for more details and examples.

Sorting

- *sort_axes*: sort the labels of an axis.
- *labelsofsorted*: give labels which would sort an axis.
- *sort_values*: sort axes according to values

```
[22]: # get a copy of the 'pop_benelux' array
pop_benelux = demo_eurostat.pop_benelux.copy()
pop_benelux

-----
NameError                                Traceback (most recent call last)
<ipython-input-22-a960c49ccebfb> in <module>
      1 # get a copy of the 'pop_benelux' array
----> 2 pop_benelux = demo_eurostat.pop_benelux.copy()
      3 pop_benelux

NameError: name 'demo_eurostat' is not defined
```

Sort an axis (alphabetically if labels are strings)

```
[23]: pop_sorted = pop_benelux.sort_axes('gender')
pop_sorted

-----
NameError                                Traceback (most recent call last)
<ipython-input-23-128d08165772> in <module>
----> 1 pop_sorted = pop_benelux.sort_axes('gender')
```

(continues on next page)

(continued from previous page)

```
2 pop_sorted
```

```
NameError: name 'pop_benelux' is not defined
```

Give labels which would sort the axis

```
[24]: pop_benelux.labelsofsorted('country')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-24-e45d81dcc459> in <module>
----> 1 pop_benelux.labelsofsorted('country')

NameError: name 'pop_benelux' is not defined
```

Sort according to values

```
[25]: pop_sorted = pop_benelux.sort_values(('Male', 2017))
pop_sorted
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-25-c4da1fa82a19> in <module>
----> 1 pop_sorted = pop_benelux.sort_values(('Male', 2017))
      2 pop_sorted

NameError: name 'pop_benelux' is not defined
```

Combining arrays

The LArray library offers several methods and functions to combine arrays:

- *insert*: inserts an array in another array along an axis
- *append*: adds an array at the end of an axis.
- *prepend*: adds an array at the beginning of an axis.
- *extend*: extends an array along an axis.
- *stack*: combines several arrays along an axis.

Insert

```
[26]: other_countries = zeros((Axis('country=Luxembourg,Netherlands'), pop.gender, pop.
    ↪time), dtype=int)
```

```
# insert new countries before 'France'
pop_new_countries = pop.insert(other_countries, before='France')
pop_new_countries
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-26-dc5de004068f> in <module>
----> 1 other_countries = zeros((Axis('country=Luxembourg,Netherlands'), pop.gender,
    ↪pop.time), dtype=int)
```

(continues on next page)

(continued from previous page)

```

2
3 # insert new countries before 'France'
4 pop_new_countries = pop.insert(other_countries, before='France')
5 pop_new_countries

```

`NameError`: name 'pop' is not defined

See *insert* for more details and examples.

Append

Append one element to an axis of an array:

```

[27]: # append data for 'Luxembourg'
pop_new = pop.append('country', pop_benelux['Luxembourg'], 'Luxembourg')
pop_new

-----
NameError                                Traceback (most recent call last)
<ipython-input-27-45d84e78fd69> in <module>
      1 # append data for 'Luxembourg'
----> 2 pop_new = pop.append('country', pop_benelux['Luxembourg'], 'Luxembourg')
      3 pop_new

NameError: name 'pop' is not defined

```

The value being appended can have missing (or even extra) axes as long as common axes are compatible:

```

[28]: pop_lux = Array([-1, 1], pop.gender)
pop_lux

-----
NameError                                Traceback (most recent call last)
<ipython-input-28-ded64bec24ce> in <module>
----> 1 pop_lux = Array([-1, 1], pop.gender)
      2 pop_lux

NameError: name 'pop' is not defined

```

```

[29]: pop_new = pop.append('country', pop_lux, 'Luxembourg')
pop_new

-----
NameError                                Traceback (most recent call last)
<ipython-input-29-d8b88e525bb0> in <module>
----> 1 pop_new = pop.append('country', pop_lux, 'Luxembourg')
      2 pop_new

NameError: name 'pop' is not defined

```

See *append* for more details and examples.

Prepend

Prepend one element to an axis of an array:

```
[30]: # append data for 'Luxembourg'
pop_new = pop.prepend('country', pop_benelux['Luxembourg'], 'Luxembourg')
pop_new

-----
NameError                                Traceback (most recent call last)
<ipython-input-30-2d1c7645ce3f> in <module>
      1 # append data for 'Luxembourg'
----> 2 pop_new = pop.prepend('country', pop_benelux['Luxembourg'], 'Luxembourg')
      3 pop_new

NameError: name 'pop' is not defined
```

See *prepend* for more details and examples.

Extend

Extend an array along an axis with another array *with* that axis (but other labels)

```
[31]: pop_extended = pop.extend('country', pop_benelux[['Luxembourg', 'Netherlands']])
pop_extended

-----
NameError                                Traceback (most recent call last)
<ipython-input-31-901cdde89f99> in <module>
----> 1 pop_extended = pop.extend('country', pop_benelux[['Luxembourg', 'Netherlands'
↪]])
      2 pop_extended

NameError: name 'pop' is not defined
```

See *extend* for more details and examples.

Stack

Stack several arrays together to create an entirely new dimension

```
[32]: # imagine you have loaded data for each country in different arrays
# (e.g. loaded from different Excel sheets)
pop_be = pop['Belgium']
pop_fr = pop['France']
pop_de = pop['Germany']

pop_stacked = stack({'Belgium': pop_be, 'France': pop_fr, 'Germany': pop_de}, 'country'
↪)
pop_stacked

-----
NameError                                Traceback (most recent call last)
<ipython-input-32-b62297997bac> in <module>
      1 # imagine you have loaded data for each country in different arrays
      2 # (e.g. loaded from different Excel sheets)
```

(continues on next page)

(continued from previous page)

```

----> 3 pop_be = pop['Belgium']
      4 pop_fr = pop['France']
      5 pop_de = pop['Germany']

NameError: name 'pop' is not defined

```

See [stack](#) for more details and examples.

Aligning Arrays

The `align` method align two arrays on their axes with a specified join method. In other words, it ensure all common axes are compatible.

```

[33]: # get a copy of the 'births' array
      births = demo_eurostat.births.copy()

      # align the two arrays with the 'inner' join method
      pop_aligned, births_aligned = pop_benelux.align(births, join='inner')

-----
NameError                                Traceback (most recent call last)
<ipython-input-33-9ac56279d5ed> in <module>
      1 # get a copy of the 'births' array
----> 2 births = demo_eurostat.births.copy()
      3
      4 # align the two arrays with the 'inner' join method
      5 pop_aligned, births_aligned = pop_benelux.align(births, join='inner')

NameError: name 'demo_eurostat' is not defined

```

```

[34]: print('pop_benelux before align:')
      print(pop_benelux)
      print()
      print('pop_benelux after align:')
      print(pop_aligned)

pop_benelux before align:

-----
NameError                                Traceback (most recent call last)
<ipython-input-34-032609b23e87> in <module>
      1 print('pop_benelux before align:')
----> 2 print(pop_benelux)
      3 print()
      4 print('pop_benelux after align:')
      5 print(pop_aligned)

NameError: name 'pop_benelux' is not defined

```

```

[35]: print('births before align:')
      print(births)
      print()
      print('births after align:')
      print(births_aligned)

```

births before align:

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-35-2a930ae5ee1a> in <module>
      1 print('births before align:')
----> 2 print(births)
      3 print()
      4 print('births after align:')
      5 print(births_aligned)

NameError: name 'births' is not defined

```

Aligned arrays can then be used in arithmetic operations:

```
[36]: pop_aligned - births_aligned
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-36-15f7c12091d3> in <module>
----> 1 pop_aligned - births_aligned

NameError: name 'pop_aligned' is not defined

```

See *align* for more details and examples.

4.2.5 Indexing, Selecting and Assigning

Import the LArray library:

```
[1]: from larray import *
```

Import the test array pop:

```
[2]: # let's start with
pop = load_example_data('demography_eurostat').pop
pop
```

```

-----
AttributeError                            Traceback (most recent call last)
<ipython-input-2-9782f2255391> in <module>
      1 # let's start with
----> 2 pop = load_example_data('demography_eurostat').pop
      3 pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
     91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
     92         raise ValueError("example_data must be chosen from list {}".
↳format(list(AVAILABLE_EXAMPLE_DATA.keys())))
--> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
     94         if isinstance(a0, str):
     95             # assume a0 is a filename
--> 96             self.load(a0)
     97         else:
     98             # iterable of tuple or dict-like

```

(continues on next page)

(continued from previous page)

```

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
    426         else:
    427             handler = handler_cls(fname)
--> 428         metadata, objects = handler.read(names, display=display, **kwargs)
    429         for k, v in objects.items():
    430             self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
    128         print("loading", type, "object", key, "...", end=' ')
    129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
    131         except Exception:
    132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
    137         else:
    138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
    140
    141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
    81         cartesian_prod = writer != 'LArray'
    82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83                             parse_header=False, cartesian_prod=cartesian_
↳ prod)
    84         if _meta is not None:
    85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '

```

(continues on next page)

(continued from previous page)

```

242                                     'Please call the method sort_axes on the_
↳returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to_
↳convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41         Returns unique labels for each dimension.
    42         """
---> 43         if isinstance(idx, pd.core.index.MultiIndex):
    44             if sort:
    45                 return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

Selecting (Subsets)

The Array class allows to select a subset either by labels or indices (positions)

Selecting by Labels

To take a subset of an array using labels, use brackets [].

Let's start by selecting a single element:

```
[3]: pop['Belgium', 'Female', 2017]
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-3-416b922b672e> in <module>
----> 1 pop['Belgium', 'Female', 2017]

NameError: name 'pop' is not defined

```

As long as there is no ambiguity (i.e. axes sharing one or several same label(s)), the order of indexing does not matter. So you usually do not care/have to remember about axes positions during computation. It only matters for output.

```
[4]: # order of index doesn't matter
pop['Female', 2017, 'Belgium']
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-4-4385ae601c5e> in <module>
      1 # order of index doesn't matter
----> 2 pop['Female', 2017, 'Belgium']

NameError: name 'pop' is not defined

```

Selecting a subset is done by using slices or lists of labels:

```
[5]: pop[['Belgium', 'Germany'], 2014:2016]
```



```
-----
NameError                                Traceback (most recent call last)
<ipython-input-5-dffb7fcae00c> in <module>
----> 1 pop[['Belgium', 'Germany'], 2014:2016]

NameError: name 'pop' is not defined
```

Slices bounds are optional: if not given, start is assumed to be the first label and stop is the last one.

```
[6]: # select all years starting from 2015
pop[2015:]
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-6-cd526a81a33e> in <module>
      1 # select all years starting from 2015
----> 2 pop[2015:]

NameError: name 'pop' is not defined
```

```
[7]: # select all first years until 2015
pop[:2015]
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-7-d85985428ba9> in <module>
      1 # select all first years until 2015
----> 2 pop[:2015]

NameError: name 'pop' is not defined
```

Slices can also have a step (defaults to 1), to take every Nth labels:

```
[8]: # select all even years starting from 2014
pop[2014::2]
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-8-2f5495b33d72> in <module>
      1 # select all even years starting from 2014
----> 2 pop[2014::2]

NameError: name 'pop' is not defined
```

Warning: Selecting by labels as in above examples works well as long as there is no ambiguity. When two or more axes have common labels, it leads to a crash. The solution is then to precise to which axis belong the labels.

```
[9]: immigration = load_example_data('demography_eurostat').immigration

# the 'immigration' array has two axes (country and citizenship) which share the same
↪ labels
immigration
```

```
-----
AttributeError                            Traceback (most recent call last)
<ipython-input-9-289b1fb5c9ce> in <module>
```

(continues on next page)

(continued from previous page)

```

----> 1 immigration = load_example_data('demography_eurostat').immigration
      2
      3 # the 'immigration' array has two axes (country and citizenship) which share_
↳the same labels
      4 immigration

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
↳format(list(AVAILABLE_EXAMPLE_DATA.keys())))
--> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
--> 96         self.load(a0)
      97     else:
      98         # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳display, **kwargs)
      426     else:
      427         handler = handler_cls(fname)
--> 428     metadata, objects = handler.read(names, display=display, **kwargs)
      429     for k, v in objects.items():
      430         self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
      128         print("loading", type, "object", key, "...", end=' ')
      129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
      131         except Exception:
      132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳**kwargs)
      137     else:
      138         raise TypeError()
--> 139     return read_hdf(self.handle, hdf_key, *args, **kwargs)
      140
      141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳value, na, sort_rows, sort_columns, name, **kwargs)
      81         cartesian_prod = writer != 'LArray'
      82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳columns, fill_value=fill_value,
--> 83                             parse_header=False, cartesian_prod=cartesian_
↳prod)
      84         if _meta is not None:
      85             res.meta = _meta

```

(continues on next page)

(continued from previous page)

```
~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳ ' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340         unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
    242         'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
    244
    245     # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'
```

```
[10]: # LArray doesn't use the position of the labels used inside the brackets
# to determine the corresponding axes. Instead LArray will try to guess the
# corresponding axis for each label whatever is its position.
# Then, if a label is shared by two or more axes, LArray will not be able
# to choose between the possible axes and will raise an error.
```

```
try:
    immigration['Belgium', 'Netherlands']
except Exception as e:
    print(type(e).__name__, ':', e)
```

```
NameError : name 'immigration' is not defined
```

```
[11]: # the solution is simple. You need to precise the axes on which you make a selection
immigration[immigration.country['Belgium'], immigration.citizenship['Netherlands']]
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-11-e9996eb15483> in <module>
      1 # the solution is simple. You need to precise the axes on which you make a
↳ selection
```

(continues on next page)

(continued from previous page)

```
----> 2 immigration[immigration.country['Belgium'], immigration.citizenship[
↳ 'Netherlands']]

NameError: name 'immigration' is not defined
```

Ambiguous Cases - Specifying Axes Using The Special Variable X

When selecting, assigning or using aggregate functions, an axis can be referred via the special variable X:

- `pop[X.time[2015:]]`
- `pop.sum(X.time)`

This gives you access to axes of the array you are manipulating. The main drawback of using X is that you lose the autocompletion available from many editors. It only works with non-anonymous axes for which names do not contain whitespaces or special characters.

```
[12]: # the previous example can also be written as
immigration[X.country['Belgium'], X.citizenship['Netherlands']]

-----
NameError                                Traceback (most recent call last)
<ipython-input-12-d269086d0278> in <module>
      1 # the previous example can also be written as
----> 2 immigration[X.country['Belgium'], X.citizenship['Netherlands']]

NameError: name 'immigration' is not defined
```

Selecting by Indices

Sometimes it is more practical to use indices (positions) along the axis, instead of labels. You need to add the character `i` before the brackets: `.i[indices]`. As for selection with labels, you can use a single index, a slice or a list of indices. Indices can be also negative (`-1` represent the last element of an axis).

Note: Remember that indices (positions) are always **0-based** in Python. So the first element is at index 0, the second is at index 1, etc.

```
[13]: # select the last year
pop[X.time.i[-1]]

-----
NameError                                Traceback (most recent call last)
<ipython-input-13-64147c3366d4> in <module>
      1 # select the last year
----> 2 pop[X.time.i[-1]]

NameError: name 'pop' is not defined
```

```
[14]: # same but for the last 3 years
pop[X.time.i[-3:]]
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-14-1174dfab392c> in <module>
      1 # same but for the last 3 years
----> 2 pop[X.time.i[-3:]]

NameError: name 'pop' is not defined

```

```
[15]: # using a list of indices
pop[X.time.i[0, 2, 4]]
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-15-db2d9cb9c286> in <module>
      1 # using a list of indices
----> 2 pop[X.time.i[0, 2, 4]]

NameError: name 'pop' is not defined

```

Warning: The end *indice* (position) is EXCLUSIVE while the end label is INCLUSIVE.

```
[16]: year = 2015
```

```

# with labels
pop[X.time[:year]]

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-16-9a77b8cef09c> in <module>
      2
      3 # with labels
----> 4 pop[X.time[:year]]

NameError: name 'pop' is not defined

```

```
[17]: # with indices (i.e. using the .i[indices] syntax)
index_year = pop.time.index(year)
pop[X.time.i[:index_year]]
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-17-ae9e14a67775> in <module>
      1 # with indices (i.e. using the .i[indices] syntax)
----> 2 index_year = pop.time.index(year)
      3 pop[X.time.i[:index_year]]

NameError: name 'pop' is not defined

```

You can use `.i[]` selection directly on array instead of axes. In this context, if you want to select a subset of the first and third axes for example, you must use a full slice `:` for the second one.

```
[18]: # select first country and last three years
pop.i[0, :, -3:]
```

```

-----
NameError                                Traceback (most recent call last)

```

(continues on next page)

(continued from previous page)

```
<ipython-input-18-c2b90aedefcee> in <module>
      1 # select first country and last three years
----> 2 pop.i[0, :, -3:]
```

```
NameError: name 'pop' is not defined
```

Using Groups In Selections

```
[19]: even_years = pop.time[2014::2]
```

```
pop[even_years]
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-19-d6091d3c0ac8> in <module>
----> 1 even_years = pop.time[2014::2]
      2
      3 pop[even_years]
```

```
NameError: name 'pop' is not defined
```

Boolean Filtering

Boolean filtering can be used to extract subsets. Filtering can be done on axes:

```
[20]: # select even years
pop[X.time % 2 == 0]
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-20-17c7fddce252> in <module>
      1 # select even years
----> 2 pop[X.time % 2 == 0]
```

```
NameError: name 'pop' is not defined
```

or data:

```
[21]: # select population for the year 2017
pop_2017 = pop[2017]

# select all data with a value greater than 30 million
pop_2017[pop_2017 > 30e6]
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-21-cfbf42b0a771> in <module>
      1 # select population for the year 2017
----> 2 pop_2017 = pop[2017]
      3
      4 # select all data with a value greater than 30 million
      5 pop_2017[pop_2017 > 30e6]
```

```
NameError: name 'pop' is not defined
```

Note: Be aware that after boolean filtering, several axes may have merged.

Arrays can also be used to create boolean filters:

```
[22]: start_year = Array([2015, 2016, 2017], axes=pop.country)
      start_year

-----
NameError                                Traceback (most recent call last)
<ipython-input-22-b9058e5232fa> in <module>
----> 1 start_year = Array([2015, 2016, 2017], axes=pop.country)
      2 start_year

NameError: name 'pop' is not defined
```

```
[23]: pop[X.time >= start_year]

-----
NameError                                Traceback (most recent call last)
<ipython-input-23-7e33b1b1e14e> in <module>
----> 1 pop[X.time >= start_year]

NameError: name 'pop' is not defined
```

Assigning subsets

Assigning A Value

Assigning a value to a subset is simple:

```
[24]: pop[2017] = 0
      pop

-----
NameError                                Traceback (most recent call last)
<ipython-input-24-1446b8540f40> in <module>
----> 1 pop[2017] = 0
      2 pop

NameError: name 'pop' is not defined
```

Now, let's store a subset in a new variable and modify it:

```
[25]: # store the data associated with the year 2016 in a new variable
      pop_2016 = pop[2016]
      pop_2016

-----
NameError                                Traceback (most recent call last)
<ipython-input-25-91ce7069d764> in <module>
      1 # store the data associated with the year 2016 in a new variable
----> 2 pop_2016 = pop[2016]
      3 pop_2016

NameError: name 'pop' is not defined
```

```
[26]: # now, we modify the new variable
pop_2016['Belgium'] = 0

# and we can see that the original array has been also modified
pop

-----
NameError                                Traceback (most recent call last)
<ipython-input-26-f6bcbld53a23> in <module>
      1 # now, we modify the new variable
----> 2 pop_2016['Belgium'] = 0
      3
      4 # and we can see that the original array has been also modified
      5 pop

NameError: name 'pop_2016' is not defined
```

One very important gotcha though...

Warning: Storing a subset of an array in a new variable and modifying it after may also impact the original array. The reason is that selecting a contiguous subset of the data does not return a copy of the selected subset, but rather a view on a subset of the array. To avoid such behavior, use the `.copy()` method.

Remember:

- taking a contiguous subset of an array is extremely fast (no data is copied)
- if one modifies that subset, one also **modifies the original array**
- `.copy()` returns a copy of the subset (takes speed and memory) but allows you to change the subset without modifying the original array in the same time

The same warning apply for entire arrays:

```
[27]: # reload the 'pop' array
pop = load_example_data('demography_eurostat').pop

# create a second 'pop2' variable
pop2 = pop
pop2

-----
AttributeError                            Traceback (most recent call last)
<ipython-input-27-1d07543f3912> in <module>
      1 # reload the 'pop' array
----> 2 pop = load_example_data('demography_eurostat').pop
      3
      4 # create a second 'pop2' variable
      5 pop2 = pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
    91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
    92         raise ValueError("example_data must be chosen from list {}".
format(list(AVAILABLE_EXAMPLE_DATA.keys())))
--> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, <continues on next page>)
```


(continued from previous page)

```

94         if isinstance(a0, str):
95             # assume a0 is a filename
--> 96             self.load(a0)
97         else:
98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
426         else:
427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
429             for k, v in objects.items():
430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
128             print("loading", type, "object", key, "...", end=' ')
129             try:
--> 130                 res[key] = self._read_item(key, type, *args, **kwargs)
131             except Exception:
132                 if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
137         else:
138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
140
141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
81         cartesian_prod = writer != 'LArray'
82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83                         parse_header=False, cartesian_prod=cartesian_
↳ prod)
84         if _meta is not None:
85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                         unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)

```

(continues on next page)

(continued from previous page)

```

241         raise ValueError('sort_rows and sort_columns cannot not be used_
↳when cartesian_prod is set to False. '
242         'Please call the method sort_axes on the_
↳returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245     # Pandas treats column labels as column names (strings) so we need to_
↳convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```

[28]: # set all data corresponding to the year 2017 to 0
pop2[2017] = 0
pop2

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-28-e8b13cd922ea> in <module>
    1 # set all data corresponding to the year 2017 to 0
----> 2 pop2[2017] = 0
    3 pop2

NameError: name 'pop2' is not defined

```

```

[29]: # and now take a look of what happened to the original array 'pop'
# after modifying the 'pop2' array
pop

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-29-269253ee2d9d> in <module>
    1 # and now take a look of what happened to the original array 'pop'
    2 # after modifying the 'pop2' array
----> 3 pop

NameError: name 'pop' is not defined

```

Warning: The syntax `new_array = old_array` does not create a new array but rather an ‘alias’ variable. To actually create a new array as a copy of a previous one, the `.copy()` method must be called.

```

[30]: # reload the 'pop' array
pop = load_example_data('demography_eurostat').pop

# copy the 'pop' array and store the copy in a new variable
pop2 = pop.copy()

```

(continues on next page)

(continued from previous page)

```

# modify the copy
pop2[2017] = 0
pop2

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-30-c76cccf49dbd> in <module>
      1 # reload the 'pop' array
----> 2 pop = load_example_data('demography_eurostat').pop
      3
      4 # copy the 'pop' array and store the copy in a new variable
      5 pop2 = pop.copy()

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
->format(list(AVAILABLE_EXAMPLE_DATA.keys())))
----> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
----> 96             self.load(a0)
      97         else:
      98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
->display, **kwargs)
     426         else:
     427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
     429             for k, v in objects.items():
     430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
     128         print("loading", type, "object", key, "...", end=' ')
     129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
     131         except Exception:
     132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
->**kwargs)
     137         else:
     138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
     140
     141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
->value, na, sort_rows, sort_columns, name, **kwargs)

```

(continues on next page)

(continued from previous page)

```

81         cartesian_prod = writer != 'LArray'
82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳columns, fill_value=fill_value,
---> 83             parse_header=False, cartesian_prod=cartesian_
↳prod)
84         if _meta is not None:
85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳parse_header=parse_header,
-> 340             unfold_last_axis_name=unfold_last_axis_name,
↳cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳**kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used
↳when cartesian_prod is set to False. '
242             'Please call the method sort_axes on the
↳returned array to sort rows or columns')
-> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to
↳convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41         Returns unique labels for each dimension.
42         """
---> 43         if isinstance(idx, pd.core.index.MultiIndex):
44             if sort:
45                 return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```
[31]: # the data from the original array have not been modified
pop
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-31-0dd2f7db30f1> in <module>
      1 # the data from the original array have not been modified
----> 2 pop

NameError: name 'pop' is not defined

```

Assigning Arrays And Broadcasting

Instead of a value, we can also assign an array to a subset. In that case, that array can have less axes than the target but those which are present must be compatible with the subset being targeted.

```
[32]: # select population for the year 2015
pop_2015 = pop[2015]

# propagate population for the year 2015 to all next years
pop[2016:] = pop_2015

pop

-----
NameError                                Traceback (most recent call last)
<ipython-input-32-a418c1132dc5> in <module>
      1 # select population for the year 2015
----> 2 pop_2015 = pop[2015]
      3
      4 # propagate population for the year 2015 to all next years
      5 pop[2016:] = pop_2015

NameError: name 'pop' is not defined
```

Warning: The array being assigned must have compatible axes (i.e. same axes names and same labels) with the target subset.

```
[33]: # replace 'Male' and 'Female' labels by 'M' and 'F'
pop_2015 = pop_2015.set_labels('gender', 'M,F')
pop_2015

-----
NameError                                Traceback (most recent call last)
<ipython-input-33-9d98e4e27eed> in <module>
      1 # replace 'Male' and 'Female' labels by 'M' and 'F'
----> 2 pop_2015 = pop_2015.set_labels('gender', 'M,F')
      3 pop_2015

NameError: name 'pop_2015' is not defined
```

```
[34]: # now let's try to repeat the assignement operation above with the new labels.
# An error is raised because of incompatible axes
try:
    pop[2016:] = pop_2015
except Exception as e:
    print(type(e).__name__, ': ', e)

NameError : name 'pop_2015' is not defined
```

4.2.6 Arithmetic Operations And Aggregations

Import the LArray library:

```
[1]: from larray import *
```

Arithmetic operations

```
[2]: arr = ndtest((3, 3))  
arr
```

```
[2]: a\b  b0  b1  b2  
     a0   0   1   2  
     a1   3   4   5  
     a2   6   7   8
```

One can do all usual arithmetic operations on an array, it will apply the operation to all elements individually

```
[3]: # addition  
arr + 10
```

```
[3]: a\b  b0  b1  b2  
     a0  10  11  12  
     a1  13  14  15  
     a2  16  17  18
```

```
[4]: # multiplication  
arr * 2
```

```
[4]: a\b  b0  b1  b2  
     a0   0   2   4  
     a1   6   8  10  
     a2  12  14  16
```

```
[5]: # 'true' division  
arr / 2
```

```
[5]: a\b   b0   b1   b2  
     a0  0.0  0.5  1.0  
     a1  1.5  2.0  2.5  
     a2  3.0  3.5  4.0
```

```
[6]: # 'floor' division  
arr // 2
```

```
[6]: a\b  b0  b1  b2  
     a0   0   0   1  
     a1   1   2   2  
     a2   3   3   4
```

Warning: Python has two different division operators:

- the ‘true’ division (`/`) always returns a float.
- the ‘floor’ division (`//`) returns an integer result (discarding any fractional result).

```
[7]: # % means modulo (aka remainder of division)
arr % 5
```

```
[7]: a\b   b0   b1   b2
      a0    0    1    2
      a1    3    4    0
      a2    1    2    3
```

```
[8]: # ** means raising to the power
arr ** 3
```

```
[8]: a\b   b0   b1   b2
      a0    0    1    8
      a1   27   64  125
      a2  216  343  512
```

More interestingly, binary operators as above also works between two arrays:

```
[9]: # load the 'demography_eurostat' dataset
demo_eurostat = load_example_data('demography_eurostat')

# extract the 'pop' array
pop = demo_eurostat.pop
pop
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-9-d8c998aa1085> in <module>
      1 # load the 'demography_eurostat' dataset
----> 2 demo_eurostat = load_example_data('demography_eurostat')
      3
      4 # extract the 'pop' array
      5 pop = demo_eurostat.pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
↳ format(list(AVAILABLE_EXAMPLE_DATA.keys())))
----> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
----> 96             self.load(a0)
      97         else:
      98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine, _
↳ display, **kwargs)
     426         else:
     427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
     429             for k, v in objects.items():
     430                 self[k] = v
```

(continues on next page)

(continued from previous page)

```

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
    128         print("loading", type, "object", key, "...", end=' ')
    129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
    131         except Exception:
    132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
    137         else:
    138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
    140
    141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
    81         cartesian_prod = writer != 'LArray'
    82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83                             parse_header=False, cartesian_prod=cartesian_
↳ prod)
    84         if _meta is not None:
    85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
    242                             'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
    244
    245         # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41         Returns unique labels for each dimension.

```

(continues on next page)

(continued from previous page)

```

42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
44         if sort:
45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```

[10]: aggregation_matrix = Array([[1, 0, 0], [0, 1, 1]], axes=(Axis('country=Belgium,
↪France+Germany'), pop.country))
aggregation_matrix

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-10-9c80f7bb843d> in <module>
--> 1 aggregation_matrix = Array([[1, 0, 0], [0, 1, 1]], axes=(Axis(
↪'country=Belgium,France+Germany'), pop.country))
      2 aggregation_matrix

NameError: name 'pop' is not defined

```

```

[11]: # @ means matrix product
aggregation_matrix @ pop['Male']

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-11-30029397c219> in <module>
      1 # @ means matrix product
--> 2 aggregation_matrix @ pop['Male']

NameError: name 'aggregation_matrix' is not defined

```

Note: Be careful when mixing different data types. You can use the method *astype* to change the data type of an array.

```

[12]: aggregation_matrix = Array([[1, 0, 0], [0, 0.5, 0.5]], axes=(Axis('country=Belgium,
↪France+Germany/2'), pop.country))
aggregation_matrix

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-12-2cda65e94db0> in <module>
--> 1 aggregation_matrix = Array([[1, 0, 0], [0, 0.5, 0.5]], axes=(Axis(
↪'country=Belgium,France+Germany/2'), pop.country))
      2 aggregation_matrix

NameError: name 'pop' is not defined

```

```

[13]: aggregation_matrix @ pop['Male']

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-13-c758434338eb> in <module>
--> 1 aggregation_matrix @ pop['Male']

NameError: name 'aggregation_matrix' is not defined

```

```
[14]: # force the resulting matrix to be an integer matrix
      (aggregation_matrix @ pop['Male']).astype(int)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-14-1298c22f1d57> in <module>
      1 # force the resulting matrix to be an integer matrix
----> 2 (aggregation_matrix @ pop['Male']).astype(int)

NameError: name 'aggregation_matrix' is not defined
```

Axis order does not matter much (except for output)

You can do operations between arrays having different axes order. The axis order of the result is the same as the left array

```
[15]: # extract the 'births' array
      births = demo_eurostat.births

      # let's change the order of axes of the 'births' array
      births_transposed = births.transpose()
      births_transposed
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-15-e465alda3e89> in <module>
      1 # extract the 'births' array
----> 2 births = demo_eurostat.births
      3
      4 # let's change the order of axes of the 'births' array
      5 births_transposed = births.transpose()

NameError: name 'demo_eurostat' is not defined
```

```
[16]: # LArray doesn't care of axes order when performing
      # arithmetic operations between arrays
      pop + births_transposed
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-16-a86edf97720f> in <module>
      1 # LArray doesn't care of axes order when performing
      2 # arithmetic operations between arrays
----> 3 pop + births_transposed

NameError: name 'pop' is not defined
```

Axes must be compatible

Arithmetic operations between two arrays only works when they have compatible axes (i.e. same labels).

```
[17]: # the 'pop' and 'births' have compatible axes
pop + births
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-17-8a09b11d5b88> in <module>
      1 # the 'pop' and 'births' have compatible axes
----> 2 pop + births

NameError: name 'pop' is not defined
```

```
[18]: # Now, let's replace the country names by the country codes
births_codes = births.set_labels('country', ['BE', 'FR', 'DE'])
births_codes
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-18-93b8c53b345d> in <module>
      1 # Now, let's replace the country names by the country codes
----> 2 births_codes = births.set_labels('country', ['BE', 'FR', 'DE'])
      3 births_codes

NameError: name 'births' is not defined
```

```
[19]: # arithmetic operations between arrays
# having incompatible axes raise an error
try:
    pop + births_codes
except Exception as e:
    print(type(e).__name__, e)
```

```
NameError name 'pop' is not defined
```

Warning: Operations between two arrays only works when they have compatible axes (i.e. same labels) but this behavior can be override via the *ignore_labels* method. In that case only the position on the axis is used and not the labels. Using this method is done at your own risk.

```
[20]: # use the .ignore_labels() method on axis 'country'
# to avoid the incompatible axes error (risky)
pop + births_codes.ignore_labels('country')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-20-4bc895ec8d89> in <module>
      1 # use the .ignore_labels() method on axis 'country'
      2 # to avoid the incompatible axes error (risky)
----> 3 pop + births_codes.ignore_labels('country')

NameError: name 'pop' is not defined
```

Extra Or Missing Axes (Broadcasting)

The condition that axes must be compatible only applies on common axes. Arithmetic operations between two arrays can be performed even if the second array has extra or missing axes compared to the first one:

```
[21]: # let's define a 'multiplier' vector with
      # one value defined for each gender
      multiplier = Array([-1, 1], axes=pop.gender)
      multiplier

-----
NameError                                Traceback (most recent call last)
<ipython-input-21-de074b9d8f5c> in <module>
      1 # let's define a 'multiplier' vector with
      2 # one value defined for each gender
----> 3 multiplier = Array([-1, 1], axes=pop.gender)
      4 multiplier

NameError: name 'pop' is not defined
```

```
[22]: # the multiplication below has been propagated to the
      # 'country' and 'time' axes.
      # This behavior is called broadcasting
      pop * multiplier

-----
NameError                                Traceback (most recent call last)
<ipython-input-22-9b1afbba6ff> in <module>
      2 # 'country' and 'time' axes.
      3 # This behavior is called broadcasting
----> 4 pop * multiplier

NameError: name 'pop' is not defined
```

Boolean Operations

Python comparison operators are:

Operator	Meaning
==	equal
!=	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

Applying a comparison operator on an array returns a boolean array:

```
[23]: # test which values are greater than 10 millions
      pop > 10e6

-----
NameError                                Traceback (most recent call last)
<ipython-input-23-641c4fe49e3a> in <module>
      1 # test which values are greater than 10 millions
```

(continues on next page)

(continued from previous page)

```
----> 2 pop > 10e6
```

```
NameError: name 'pop' is not defined
```

Comparison operations can be combined using Python bitwise operators:

Operator	Meaning
&	and
	or
~	not

```
[24]: # test which values are greater than 10 millions and less than 40 millions
      (pop > 10e6) & (pop < 40e6)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-24-229e41e4b2b3> in <module>
      1 # test which values are greater than 10 millions and less than 40 millions
----> 2 (pop > 10e6) & (pop < 40e6)

NameError: name 'pop' is not defined
```

```
[25]: # test which values are less than 10 millions or greater than 40 millions
      (pop < 10e6) | (pop > 40e6)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-25-d287c6a8a6e5> in <module>
      1 # test which values are less than 10 millions or greater than 40 millions
----> 2 (pop < 10e6) | (pop > 40e6)

NameError: name 'pop' is not defined
```

```
[26]: # test which values are not less than 10 millions
      ~(pop < 10e6)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-26-511032bcaa94> in <module>
      1 # test which values are not less than 10 millions
----> 2 ~(pop < 10e6)

NameError: name 'pop' is not defined
```

The returned boolean array can then be used in selections and assignments:

```
[27]: pop_copy = pop.copy()

      # set all values greater than 40 millions to 40 millions
      pop_copy[pop_copy > 40e6] = 40e6
      pop_copy
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-27-5c28ad882397> in <module>
----> 1 pop_copy = pop.copy()
```

(continues on next page)

(continued from previous page)

```
2
3 # set all values greater than 40 millions to 40 millions
4 pop_copy[pop_copy > 40e6] = 40e6
5 pop_copy
```

```
NameError: name 'pop' is not defined
```

Boolean operations can be made between arrays:

```
[28]: # test where the two arrays have the same values
pop == pop_copy
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-28-bd6bc78c0981> in <module>
      1 # test where the two arrays have the same values
----> 2 pop == pop_copy

NameError: name 'pop' is not defined
```

To test if all values between are equals, use the *equals* method:

```
[29]: pop.equals(pop_copy)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-29-19ebac2262e0> in <module>
----> 1 pop.equals(pop_copy)

NameError: name 'pop' is not defined
```

Aggregates

The LArray library provides many aggregation functions. The list is given in the *Aggregation Functions* subsection of the *API Reference* page.

Aggregation operations can be performed on axes or groups. Axes and groups can be mixed.

The main rules are:

- Axes are separated by commas ,
- Groups belonging to the same axis are grouped inside parentheses ()

Calculate the sum along an axis:

```
[30]: pop.sum('gender')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-30-69a08e55585e> in <module>
----> 1 pop.sum('gender')

NameError: name 'pop' is not defined
```

or several axes (axes are separated by commas ,):

```
[31]: pop.sum('country', 'gender')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-31-90acf6310fef> in <module>
----> 1 pop.sum('country', 'gender')

NameError: name 'pop' is not defined
```

Calculate the sum along all axes except one by appending `_by` to the aggregation function:

```
[32]: pop.sum_by('time')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-32-fa0331e295de> in <module>
----> 1 pop.sum_by('time')

NameError: name 'pop' is not defined
```

Calculate the sum along groups (the groups belonging to the same axis must grouped inside parentheses ()):

```
[33]: even_years = pop.time[2014::2] >> 'even_years'
odd_years = pop.time[2013::2] >> 'odd_years'

pop.sum((odd_years, even_years))
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-33-6da7c15fbbb2> in <module>
----> 1 even_years = pop.time[2014::2] >> 'even_years'
      2 odd_years = pop.time[2013::2] >> 'odd_years'
      3
      4 pop.sum((odd_years, even_years))

NameError: name 'pop' is not defined
```

Mixing axes and groups in aggregations:

```
[34]: pop.sum('gender', (odd_years, even_years))
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-34-260c0b07b806> in <module>
----> 1 pop.sum('gender', (odd_years, even_years))

NameError: name 'pop' is not defined
```

4.2.7 Pythonic VS String Syntax

Import the LArray library:

```
[1]: from larray import *
```

The LArray library offers two syntaxes to build axes and make selections and aggregations. The first one is more Pythonic (uses Python structures) For example, you can create an `age_category` axis as follows:

```
[2]: age_category = Axis(["0-9", "10-17", "18-66", "67+"], "age_category")
age_category
```

```
[2]: Axis(['0-9', '10-17', '18-66', '67+'], 'age_category')
```

The second one consists of using strings that are parsed. It is shorter to type. The same *age_category* axis could have been generated as follows:

```
[3]: age_category = Axis("age_category=0-9,10-17,18-66,67+")
age_category
```

```
[3]: Axis(['0-9', '10-17', '18-66', '67+'], 'age_category')
```

Warning: The drawback of the string syntax is that some characters such as `,` `;` `=` `:` `..` `[]` `>>` have a special meaning and cannot be used with the `String` syntax. If you need to work with labels containing such special characters (when importing data from an external source for example), you have to use the `Pythonic` syntax which allows to use any character in labels.

String Syntax

Axes And Arrays creation

The string syntax allows to easily create axes.

When creating one axis, the labels are separated using `,` :

```
[4]: a = Axis('a=a0,a1,a2,a3')
a
```

```
[4]: Axis(['a0', 'a1', 'a2', 'a3'], 'a')
```

The special syntax `start..stop` generates a sequence of labels:

```
[5]: a = Axis('a=a0..a3')
a
```

```
[5]: Axis(['a0', 'a1', 'a2', 'a3'], 'a')
```

When creating an array, it is possible to define several axes in the same string using `;`

```
[6]: arr = zeros("a=a0..a2; b=b0,b1; c=c0..c5")
arr
```

```
[6]:  a  b\c   c0   c1   c2   c3   c4   c5
a0   b0  0.0  0.0  0.0  0.0  0.0  0.0
a0   b1  0.0  0.0  0.0  0.0  0.0  0.0
a1   b0  0.0  0.0  0.0  0.0  0.0  0.0
a1   b1  0.0  0.0  0.0  0.0  0.0  0.0
a2   b0  0.0  0.0  0.0  0.0  0.0  0.0
a2   b1  0.0  0.0  0.0  0.0  0.0  0.0
```


Selection

Starting from the array:

```
[7]: immigration = load_example_data('demography_eurostat').immigration
immigration.info

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-7-4d243c9b8788> in <module>
----> 1 immigration = load_example_data('demography_eurostat').immigration
      2 immigration.info

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
↳ format(list(AVAILABLE_EXAMPLE_DATA.keys())))
--> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
--> 96             self.load(a0)
      97         else:
      98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
     426         else:
     427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
     429             for k, v in objects.items():
     430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
     128         print("loading", type, "object", key, "...", end=' ')
     129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
     131         except Exception:
     132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
     137         else:
     138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
     140
     141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
      81         cartesian_prod = writer != 'LArray'
```

(continues on next page)

(continued from previous page)

```

82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83         parse_header=False, cartesian_prod=cartesian_
↳ prod)
84         if _meta is not None:
85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340         unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
242         'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41         Returns unique labels for each dimension.
42         """
--> 43         if isinstance(idx, pd.core.index.MultiIndex):
44             if sort:
45                 return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

an example of a selection using the Pythonic syntax is:

```

[8]: # since the labels 'Belgium' and 'Netherlands' also exists in the 'citizenship' axis,
# we need to explicitly specify that we want to make a selection over the 'country'
↳ axis
immigration_subset = immigration[X.country['Belgium', 'Netherlands'], 'Female', 2015:]
immigration_subset

-----
NameError                                Traceback (most recent call last)
<ipython-input-8-d9ca54fdd4a1> in <module>
      1 # since the labels 'Belgium' and 'Netherlands' also exists in the 'citizenship
↳ ' axis,
      2 # we need to explicitly specify that we want to make a selection over the
↳ 'country' axis

```

(continues on next page)

(continued from previous page)

```

----> 3 immigration_subset = immigration[X.country['Belgium', 'Netherlands'], 'Female'
↳', 2015:]
      4 immigration_subset

```

NameError: name 'immigration' is not defined

Using the String syntax, the same selection becomes:

```
[9]: immigration_subset = immigration['country[Belgium,Netherlands]', 'Female', 2015:]
immigration_subset
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-9-6d463abc555a> in <module>
----> 1 immigration_subset = immigration['country[Belgium,Netherlands]', 'Female',
↳2015:]
      2 immigration_subset

NameError: name 'immigration' is not defined

```

Aggregation

An example of an aggregation using the Pythonic syntax is:

```
[10]: immigration.sum((X.time[2014::2] >> 'even_years', X.time[:,2] >> 'odd_years'),
↳'citizenship')
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-10-d0659d645eb8> in <module>
----> 1 immigration.sum((X.time[2014::2] >> 'even_years', X.time[:,2] >> 'odd_years'),
↳ 'citizenship')

NameError: name 'immigration' is not defined

```

Using the String syntax, the same aggregation becomes:

```
[11]: immigration.sum('time[2014::2] >> even_years; time[:,2] >> odd_years', 'citizenship')
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-11-ca658d11b0fe> in <module>
----> 1 immigration.sum('time[2014::2] >> even_years; time[:,2] >> odd_years',
↳ 'citizenship')

NameError: name 'immigration' is not defined

```

where we used ; to separate groups of labels from the same axis.

4.2.8 Plotting

Import the LArray library:

```
[1]: from larray import *
```

Import a subset of the test array pop:

```
[2]: # load 'demography_eurostat' dataset
demo_eurostat = load_example_data('demography_eurostat')

# extract the 'pop' array from the dataset
pop = demo_eurostat.pop
pop
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-2-ce337d55681e> in <module>
      1 # load 'demography_eurostat' dataset
----> 2 demo_eurostat = load_example_data('demography_eurostat')
      3
      4 # extract the 'pop' array from the dataset
      5 pop = demo_eurostat.pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
↳format(list(AVAILABLE_EXAMPLE_DATA.keys())))
----> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
----> 96             self.load(a0)
      97         else:
      98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳display, **kwargs)
     426         else:
     427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
     429             for k, v in objects.items():
     430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
     128             print("loading", type, "object", key, "...", end=' ')
     129             try:
--> 130                 res[key] = self._read_item(key, type, *args, **kwargs)
     131             except Exception:
     132                 if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳**kwargs)
```

(continues on next page)

(continued from previous page)

```

137         else:
138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
140
141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
    81         cartesian_prod = writer != 'LArray'
    82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83         parse_header=False, cartesian_prod=cartesian_
↳ prod)
    84         if _meta is not None:
    85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳ ' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340         unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
    242         'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
    244
    245         # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41         Returns unique labels for each dimension.
    42         """
--> 43         if isinstance(idx, pd.core.index.MultiIndex):
    44             if sort:
    45                 return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

Inline matplotlib (required in notebooks):

```
[3]: %matplotlib inline
```

Create a plot (last axis define the different curves to draw):

```
[4]: pop.plot()
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-38bee61ed13b> in <module>
----> 1 pop.plot()

NameError: name 'pop' is not defined
```

See [plot](#) for more details and examples.

4.2.9 Some Useful Functions

Import the LArray library:

```
[1]: from larray import *
```

```
[2]: # load 'demography_eurostat' dataset
demo_eurostat = load_example_data('demography_eurostat')

# extract the 'pop' array from the dataset
pop = demo_eurostat.pop
pop
```

```
-----
AttributeError                            Traceback (most recent call last)
<ipython-input-2-ce337d55681e> in <module>
      1 # load 'demography_eurostat' dataset
----> 2 demo_eurostat = load_example_data('demography_eurostat')
      3
      4 # extract the 'pop' array from the dataset
      5 pop = demo_eurostat.pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
↳format(list(AVAILABLE_EXAMPLE_DATA.keys())))
--> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
--> 96             self.load(a0)
      97         else:
      98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳display, **kwargs)
     426         else:
```

(continues on next page)

(continued from previous page)

```

427         handler = handler_cls(fname)
--> 428     metadata, objects = handler.read(names, display=display, **kwargs)
429     for k, v in objects.items():
430         self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
128         print("loading", type, "object", key, "...", end=' ')
129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
131         except Exception:
132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
137         else:
138             raise TypeError()
--> 139         return read_hdf(self.handle, hdf_key, *args, **kwargs)
140
141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
81         cartesian_prod = writer != 'LArray'
82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83             parse_header=False, cartesian_prod=cartesian_
↳ prod)
84         if _meta is not None:
85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used_
↳ when cartesian_prod is set to False. '
242             'Please call the method sort_axes on the_
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to_
↳ convert them to values

```

(continues on next page)

(continued from previous page)

```
~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idx, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'
```

with total

Add totals to one or several axes:

```
[3]: pop.with_total('gender', label='Total')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-d5ae827e7ef1> in <module>
--> 1 pop.with_total('gender', label='Total')

NameError: name 'pop' is not defined
```

See *with_total* for more details and examples.

where

The where function can be used to apply some computation depending on a condition:

```
[4]: # where(condition, value if true, value if false)
where(pop < pop.mean('time'), -pop, pop)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-05a575f73c29> in <module>
    1 # where(condition, value if true, value if false)
--> 2 where(pop < pop.mean('time'), -pop, pop)

NameError: name 'pop' is not defined
```

See *where* for more details and examples.

clip

Set all data between a certain range:

```
[5]: # values below 10 millions are set to 10 millions
pop.clip(minval=10**7)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-5-13053b977351> in <module>
    1 # values below 10 millions are set to 10 millions
```

(continues on next page)

(continued from previous page)

```
----> 2 pop.clip(minval=10**7)

NameError: name 'pop' is not defined
```

```
[6]: # values above 40 millions are set to 40 millions
pop.clip(maxval=4*10**7)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-6-e0b0b4c8bb9c> in <module>
      1 # values above 40 millions are set to 40 millions
----> 2 pop.clip(maxval=4*10**7)

NameError: name 'pop' is not defined
```

```
[7]: # values below 10 millions are set to 10 millions and
# values above 40 millions are set to 40 millions
pop.clip(10**7, 4*10**7)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-7-ba0330376a84> in <module>
      1 # values below 10 millions are set to 10 millions and
      2 # values above 40 millions are set to 40 millions
----> 3 pop.clip(10**7, 4*10**7)

NameError: name 'pop' is not defined
```

See [clip](#) for more details and examples.

divnot0

Replace division by 0 by 0:

```
[8]: divisor = ones(pop.axes, dtype=int)
divisor['Male'] = 0
divisor
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-8-4dc88d9626bf> in <module>
----> 1 divisor = ones(pop.axes, dtype=int)
      2 divisor['Male'] = 0
      3 divisor

NameError: name 'pop' is not defined
```

```
[9]: pop / divisor
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-9-75d7cd731c57> in <module>
----> 1 pop / divisor

NameError: name 'pop' is not defined
```

```
[10]: # we use astype(int) since the divnot0 method
      # returns a float array in this case while
      # we want an integer array
      pop.divnot0(divisor).astype(int)

-----
NameError                                Traceback (most recent call last)
<ipython-input-10-864c0d271282> in <module>
      2 # returns a float array in this case while
      3 # we want an integer array
----> 4 pop.divnot0(divisor).astype(int)

NameError: name 'pop' is not defined
```

See *divnot0* for more details and examples.

ratio

The *ratio* (*rationot0*) method returns an array with all values divided by the sum of values along given axes:

```
[11]: pop.ratio('gender')

      # which is equivalent to
      pop / pop.sum('gender')

-----
NameError                                Traceback (most recent call last)
<ipython-input-11-0468caf179b2> in <module>
----> 1 pop.ratio('gender')
      2
      3 # which is equivalent to
      4 pop / pop.sum('gender')

NameError: name 'pop' is not defined
```

See *ratio* and *rationot0* for more details and examples.

percents

```
[12]: # or, if you want the previous ratios in percents
      pop.percent('gender')

-----
NameError                                Traceback (most recent call last)
<ipython-input-12-25393219a8df> in <module>
      1 # or, if you want the previous ratios in percents
----> 2 pop.percent('gender')

NameError: name 'pop' is not defined
```

See *percent* for more details and examples.

diff

The `diff` method calculates the n -th order discrete difference along a given axis.

The first order difference is given by $\text{out}[n+1] = \text{in}[n+1] - \text{in}[n]$ along the given axis.

```
[13]: # calculates 'diff[year+1] = pop[year+1] - pop[year]'
      pop.diff('time')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-13-8de85c6279ec> in <module>
      1 # calculates 'diff[year+1] = pop[year+1] - pop[year]'
----> 2 pop.diff('time')
```

```
NameError: name 'pop' is not defined
```

```
[14]: # calculates 'diff[year+2] = pop[year+2] - pop[year]'
      pop.diff('time', d=2)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-14-52d4a8a02e05> in <module>
      1 # calculates 'diff[year+2] = pop[year+2] - pop[year]'
----> 2 pop.diff('time', d=2)
```

```
NameError: name 'pop' is not defined
```

```
[15]: # calculates 'diff[year] = pop[year+1] - pop[year]'
      pop.diff('time', label='lower')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-15-ac9f0715f28c> in <module>
      1 # calculates 'diff[year] = pop[year+1] - pop[year]'
----> 2 pop.diff('time', label='lower')
```

```
NameError: name 'pop' is not defined
```

See *diff* for more details and examples.

growth_rate

The `growth_rate` method calculates the growth along a given axis.

It is roughly equivalent to `a.diff(axis, d, label) / a[axis.i[:-d]]`:

```
[16]: pop.growth_rate('time')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-16-745057b9829b> in <module>
----> 1 pop.growth_rate('time')
```

```
NameError: name 'pop' is not defined
```

See *growth_rate* for more details and examples.

shift

The `shift` method drops first label of an axis and shifts all subsequent labels

```
[17]: pop.shift('time')

-----
NameError                                Traceback (most recent call last)
<ipython-input-17-f61f4cde41a9> in <module>
----> 1 pop.shift('time')

NameError: name 'pop' is not defined
```

```
[18]: # when shift is applied on an (increasing) time axis,
# it effectively brings "past" data into the future
pop_shifted = pop.shift('time')
stack({'pop_shifted_2014': pop_shifted[2014], 'pop_2013': pop[2013]}, 'array')

-----
NameError                                Traceback (most recent call last)
<ipython-input-18-c18dcab698a8> in <module>
      1 # when shift is applied on an (increasing) time axis,
      2 # it effectively brings "past" data into the future
----> 3 pop_shifted = pop.shift('time')
      4 stack({'pop_shifted_2014': pop_shifted[2014], 'pop_2013': pop[2013]}, 'array')

NameError: name 'pop' is not defined
```

See *shift* for more details and examples.

Other interesting functions

There are a lot more interesting functions that you can find in the API reference in sections *Aggregation Functions*, *Miscellaneous* and *Utility Functions*.

4.2.10 Working With Sessions

Import the LArray library:

```
[1]: from larray import *
```

Before To Continue

If you not yet comfortable with creating, saving and loading sessions, please read first the *Creating Sessions* and *Loading and Dumping Sessions* sections of the tutorial before going further.

Exploring Content

To get the list of items names of a session, use the *names* shortcut (be careful that the list is sorted alphabetically and does not follow the internal order!):

```
[2]: # load a session representing the results of a demographic model
filepath_hdf = get_example_filepath('demography_eurostat.h5')
s_pop = Session(filepath_hdf)

# print the content of the session
print(s_pop.names)
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-2-6d556d84191c> in <module>
      1 # load a session representing the results of a demographic model
      2 filepath_hdf = get_example_filepath('demography_eurostat.h5')
----> 3 s_pop = Session(filepath_hdf)
      4
      5 # print the content of the session

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
    94         if isinstance(a0, str):
    95             # assume a0 is a filename
--> 96         self.load(a0)
    97     else:
    98         # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
    426     else:
    427         handler = handler_cls(fname)
--> 428     metadata, objects = handler.read(names, display=display, **kwargs)
    429     for k, v in objects.items():
    430         self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
    128         print("loading", type, "object", key, "...", end=' ')
    129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
    131         except Exception:
    132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
    137     else:
    138         raise TypeError()
--> 139     return read_hdf(self.handle, hdf_key, *args, **kwargs)
    140
    141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
```

(continues on next page)

(continued from previous page)

```

81         cartesian_prod = writer != 'LArray'
82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳columns, fill_value=fill_value,
---> 83             parse_header=False, cartesian_prod=cartesian_
↳prod)
84         if _meta is not None:
85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳columns, raw, parse_header, wide, cartesian_prod, **kwargs)
338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳' in axes_names[-1]
339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳parse_header=parse_header,
-> 340             unfold_last_axis_name=unfold_last_axis_name,
↳cartesian_prod=cartesian_prod, **kwargs)
341
342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳**kwargs)
241         raise ValueError('sort_rows and sort_columns cannot not be used
↳when cartesian_prod is set to False. '
242             'Please call the method sort_axes on the
↳returned array to sort rows or columns')
-> 243         axes_labels = index_to_labels(df.index, sort=False)
244
245         # Pandas treats column labels as column names (strings) so we need to
↳convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
41         Returns unique labels for each dimension.
42         """
---> 43         if isinstance(idx, pd.core.index.MultiIndex):
44             if sort:
45                 return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

To get more information of items of a session, the *summary* will provide not only the names of items but also the list of labels in the case of axes or groups and the list of axes, the shape and the dtype in the case of arrays:

```
[3]: # print the content of the session
print(s_pop.summary())
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-3-aadafdb11856> in <module>
      1 # print the content of the session
----> 2 print(s_pop.summary())

NameError: name 's_pop' is not defined

```

Selecting And Filtering Items

Session objects work like ordinary dict Python objects. To select an item, use the usual syntax `<session_var>['<item_name>']`:

```
[4]: s_pop['pop']

-----
NameError                                Traceback (most recent call last)
<ipython-input-4-c28bdf7ad79c> in <module>
----> 1 s_pop['pop']

NameError: name 's_pop' is not defined
```

A simpler way consists in the use the syntax `<session_var>.<item_name>`:

```
[5]: s_pop.pop

-----
NameError                                Traceback (most recent call last)
<ipython-input-5-e889592ecf82> in <module>
----> 1 s_pop.pop

NameError: name 's_pop' is not defined
```

Warning: The syntax `session_var.item_name` will work as long as you don't use any special character like `,` `;` `:` in the item's name.

To return a new session with selected items, use the syntax `<session_var>[list, of, item, names]`:

```
[6]: s_pop_new = s_pop['pop', 'births', 'deaths']

s_pop_new.names

-----
NameError                                Traceback (most recent call last)
<ipython-input-6-68de9e1a10e0> in <module>
----> 1 s_pop_new = s_pop['pop', 'births', 'deaths']
      2
      3 s_pop_new.names

NameError: name 's_pop' is not defined
```

The *filter* method allows you to select all items of the same kind (i.e. all axes, or groups or arrays) or all items with names satisfying a given pattern:

```
[7]: # select only arrays of a session
s_pop.filter(kind=Array)

-----
NameError                                Traceback (most recent call last)
<ipython-input-7-7af5f7f7d26e> in <module>
      1 # select only arrays of a session
----> 2 s_pop.filter(kind=Array)

NameError: name 's_pop' is not defined
```

```
[8]: # selection all items with a name starting with a letter between a and k
s_pop.filter(pattern='[a-k]*')

-----
NameError                                Traceback (most recent call last)
<ipython-input-8-a97970540334> in <module>
      1 # selection all items with a name starting with a letter between a and k
----> 2 s_pop.filter(pattern='[a-k]*')

NameError: name 's_pop' is not defined
```

Iterating over Items

Like the built-in Python dict objects, Session objects provide methods to iterate over items:

```
[9]: # iterate over item names
for key in s_pop.keys():
    print(key)

-----
NameError                                Traceback (most recent call last)
<ipython-input-9-88e5eb00e4f8> in <module>
      1 # iterate over item names
----> 2 for key in s_pop.keys():
      3     print(key)

NameError: name 's_pop' is not defined
```

```
[10]: # iterate over items
for value in s_pop.values():
    if isinstance(value, Array):
        print(value.info)
    else:
        print(repr(value))
    print()

-----
NameError                                Traceback (most recent call last)
<ipython-input-10-3bf2c61ed9b3> in <module>
      1 # iterate over items
----> 2 for value in s_pop.values():
      3     if isinstance(value, Array):
      4         print(value.info)
      5     else:

NameError: name 's_pop' is not defined
```

```
[11]: # iterate over names and items
for key, value in s_pop.items():
    if isinstance(value, Array):
        print(key, ':')
        print(value.info)
    else:
        print(key, ':', repr(value))
    print()
```



```

-----
NameError                                Traceback (most recent call last)
<ipython-input-11-cd16090b2eff> in <module>
      1 # iterate over names and items
----> 2 for key, value in s_pop.items():
      3     if isinstance(value, Array):
      4         print(key, ':')
      5         print(value.info)

NameError: name 's_pop' is not defined

```

Arithmetic Operations On Sessions

Session objects accept binary operations with a scalar:

```

[12]: # get population, births and deaths in millions
s_pop_div = s_pop / 1e6

s_pop_div.pop

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-12-70f958e0d076> in <module>
      1 # get population, births and deaths in millions
----> 2 s_pop_div = s_pop / 1e6
      3
      4 s_pop_div.pop

NameError: name 's_pop' is not defined

```

with an array (please read the documentation of the *random.choice* function first if you don't know it):

```

[13]: from larray import random
random_increment = random.choice([-1, 0, 1], p=[0.3, 0.4, 0.3], axes=s_pop.pop.axes)
↳ * 1000
random_increment

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-13-dd062de4e6dc> in <module>
      1 from larray import random
----> 2 random_increment = random.choice([-1, 0, 1], p=[0.3, 0.4, 0.3], axes=s_pop.
↳ pop.axes) * 1000
      3 random_increment

NameError: name 's_pop' is not defined

```

```

[14]: # add some variables of a session by a common array
s_pop_rand = s_pop['pop', 'births', 'deaths'] + random_increment

s_pop_rand.pop

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-14-2efaa60f64b2> in <module>
      1 # add some variables of a session by a common array
----> 2 s_pop_rand = s_pop['pop', 'births', 'deaths'] + random_increment

```

(continues on next page)

(continued from previous page)

```
3
4 s_pop_rand.pop

NameError: name 's_pop' is not defined
```

with another session:

```
[15]: # compute the difference between each array of the two sessions
s_diff = s_pop - s_pop_rand

s_diff.births

-----
NameError                                Traceback (most recent call last)
<ipython-input-15-db5241167ae2> in <module>
      1 # compute the difference between each array of the two sessions
----> 2 s_diff = s_pop - s_pop_rand
      3
      4 s_diff.births

NameError: name 's_pop' is not defined
```

Applying Functions On All Arrays

In addition to the classical arithmetic operations, the *apply* method can be used to apply the same function on all arrays. This function should take a single element argument and return a single value:

```
[16]: # add the next year to all arrays
def add_next_year(array):
    if 'time' in array.axes.names:
        last_year = array.time.i[-1]
        return array.append('time', 0, last_year + 1)
    else:
        return array

s_pop_with_next_year = s_pop.apply(add_next_year)

print('pop array before calling apply:')
print(s_pop.pop)
print()
print('pop array after calling apply:')
print(s_pop_with_next_year.pop)

-----
NameError                                Traceback (most recent call last)
<ipython-input-16-edc573de51c2> in <module>
      7         return array
      8
----> 9 s_pop_with_next_year = s_pop.apply(add_next_year)
     10
     11 print('pop array before calling apply:')

NameError: name 's_pop' is not defined
```

It is possible to pass a function with additional arguments:

```
[17]: # add the next year to all arrays.
# Use the 'copy_values_from_last_year flag' to indicate
# whether or not to copy values from the last year
def add_next_year(array, copy_values_from_last_year):
    if 'time' in array.axes.names:
        last_year = array.time.i[-1]
        value = array[last_year] if copy_values_from_last_year else 0
        return array.append('time', value, last_year + 1)
    else:
        return array

s_pop_with_next_year = s_pop.apply(add_next_year, True)

print('pop array before calling apply:')
print(s_pop.pop)
print()
print('pop array after calling apply:')
print(s_pop_with_next_year.pop)

-----
NameError                                Traceback (most recent call last)
<ipython-input-17-919adc40667d> in <module>
     10         return array
     11
--> 12 s_pop_with_next_year = s_pop.apply(add_next_year, True)
     13
     14 print('pop array before calling apply:')

NameError: name 's_pop' is not defined
```

It is also possible to apply a function on non-Array objects of a session. Please refer the documentation of the [apply](#) method.

Comparing Sessions

Being able to compare two sessions may be useful when you want to compare two different models expected to give the same results or when you have updated your model and want to see what are the consequences of the recent changes.

Session objects provide the two methods to compare two sessions: *equals* and *element_equals*:

- The `equals` method will return `True` if **all items** from both sessions are identical, `False` otherwise.
- The `element_equals` method will compare items of two sessions one by one and return an array of boolean values.

```
[18]: # load a session representing the results of a demographic model
filepath_hdf = get_example_filepath('demography_eurostat.h5')
s_pop = Session(filepath_hdf)

# create a copy of the original session
s_pop_copy = s_pop.copy()

-----
AttributeError                            Traceback (most recent call last)
<ipython-input-18-e82a48f3fb0f> in <module>
      1 # load a session representing the results of a demographic model
      2 filepath_hdf = get_example_filepath('demography_eurostat.h5')
```

(continues on next page)

(continued from previous page)

```

----> 3 s_pop = Session(filepath_hdf)
      4
      5 # create a copy of the original session

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
--> 96         self.load(a0)
      97     else:
      98         # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
     426     else:
     427         handler = handler_cls(fname)
--> 428         metadata, objects = handler.read(names, display=display, **kwargs)
     429         for k, v in objects.items():
     430             self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
     128         print("loading", type, "object", key, "...", end=' ')
     129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
     131         except Exception:
     132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
     137     else:
     138         raise TypeError()
--> 139     return read_hdf(self.handle, hdf_key, *args, **kwargs)
     140
     141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
     81         cartesian_prod = writer != 'LArray'
     82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83                             parse_header=False, cartesian_prod=cartesian_
↳ prod)
     84         if _meta is not None:
     85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
     338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\
↳ ' in axes_names[-1]
     339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340                             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
     341

```

(continues on next page)

(continued from previous page)

```

342     # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:0'
    ↪x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
    ↪columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
    ↪**kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used
    ↪when cartesian_prod is set to False. '
    242         'Please call the method sort_axes on the
    ↪returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
    244
    245     # Pandas treats column labels as column names (strings) so we need to
    ↪convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
    ↪larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idxs, sort)
    41     Returns unique labels for each dimension.
    42     """
--> 43     if isinstance(idxs, pd.core.index.MultiIndex):
    44         if sort:
    45             return list(idxs.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```

[19]: # 'element_equals' compare arrays one by one
s_pop.element_equals(s_pop_copy)

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-19-7785a94587dd> in <module>
      1 # 'element_equals' compare arrays one by one
----> 2 s_pop.element_equals(s_pop_copy)

NameError: name 's_pop' is not defined

```

```

[20]: # 'equals' returns True if all items of the two sessions have exactly the same items
s_pop.equals(s_pop_copy)

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-20-deeccf2589e2> in <module>
      1 # 'equals' returns True if all items of the two sessions have exactly the_
    ↪same items
----> 2 s_pop.equals(s_pop_copy)

NameError: name 's_pop' is not defined

```

```

[21]: # slightly modify the 'pop' array for some labels combination
s_pop_copy.pop += random_increment

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-21-01bcf10dfc4d> in <module>
      1 # slightly modify the 'pop' array for some labels combination
----> 2 s_pop_copy.pop += random_increment

```

(continues on next page)

(continued from previous page)

```
NameError: name 's_pop_copy' is not defined
```

```
[22]: # the 'pop' array is different between the two sessions
s_pop.element_equals(s_pop_copy)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-22-53b7a4e33712> in <module>
      1 # the 'pop' array is different between the two sessions
----> 2 s_pop.element_equals(s_pop_copy)

NameError: name 's_pop' is not defined
```

```
[23]: # 'equals' returns False if at least one item of the two sessions are different in_
      ↪ values or axes
s_pop.equals(s_pop_copy)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-23-1672284a9f98> in <module>
      1 # 'equals' returns False if at least one item of the two sessions are_
      ↪ different in values or axes
----> 2 s_pop.equals(s_pop_copy)

NameError: name 's_pop' is not defined
```

```
[24]: # reset the 'copy' session as a copy of the original session
s_pop_copy = s_pop.copy()

# add an array to the 'copy' session
s_pop_copy.gender_ratio = s_pop_copy.pop_ratio('gender')
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-24-c21041cc8798> in <module>
      1 # reset the 'copy' session as a copy of the original session
----> 2 s_pop_copy = s_pop.copy()
      3
      4 # add an array to the 'copy' session
      5 s_pop_copy.gender_ratio = s_pop_copy.pop_ratio('gender')

NameError: name 's_pop' is not defined
```

```
[25]: # the 'gender_ratio' array is not present in the original session
s_pop.element_equals(s_pop_copy)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-25-08dablcde8cb> in <module>
      1 # the 'gender_ratio' array is not present in the original session
----> 2 s_pop.element_equals(s_pop_copy)

NameError: name 's_pop' is not defined
```

```
[26]: # 'equals' returns False if at least one item is not present in the two sessions
s_pop.equals(s_pop_copy)
```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-26-c3385740b3c6> in <module>
      1 # 'equals' returns False if at least one item is not present in the two_
    ↪ sessions
----> 2 s_pop.equals(s_pop_copy)

NameError: name 's_pop' is not defined

```

The == operator return a new session with boolean arrays with elements compared element-wise:

```

[27]: # reset the 'copy' session as a copy of the original session
s_pop_copy = s_pop.copy()

# slightly modify the 'pop' array for some labels combination
s_pop_copy.pop += random_increment

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-27-8bcf7020c24c> in <module>
      1 # reset the 'copy' session as a copy of the original session
----> 2 s_pop_copy = s_pop.copy()
      3
      4 # slightly modify the 'pop' array for some labels combination
      5 s_pop_copy.pop += random_increment

NameError: name 's_pop' is not defined

```

```

[28]: s_check_same_values = s_pop == s_pop_copy

s_check_same_values.pop

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-28-5fd1ba111955> in <module>
----> 1 s_check_same_values = s_pop == s_pop_copy
      2
      3 s_check_same_values.pop

NameError: name 's_pop' is not defined

```

This also works for axes and groups:

```

[29]: s_check_same_values.time

```

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-29-7558bc7abc67> in <module>
----> 1 s_check_same_values.time

NameError: name 's_check_same_values' is not defined

```

The != operator does the opposite of == operator:

```

[30]: s_check_different_values = s_pop != s_pop_copy

s_check_different_values.pop

```

```

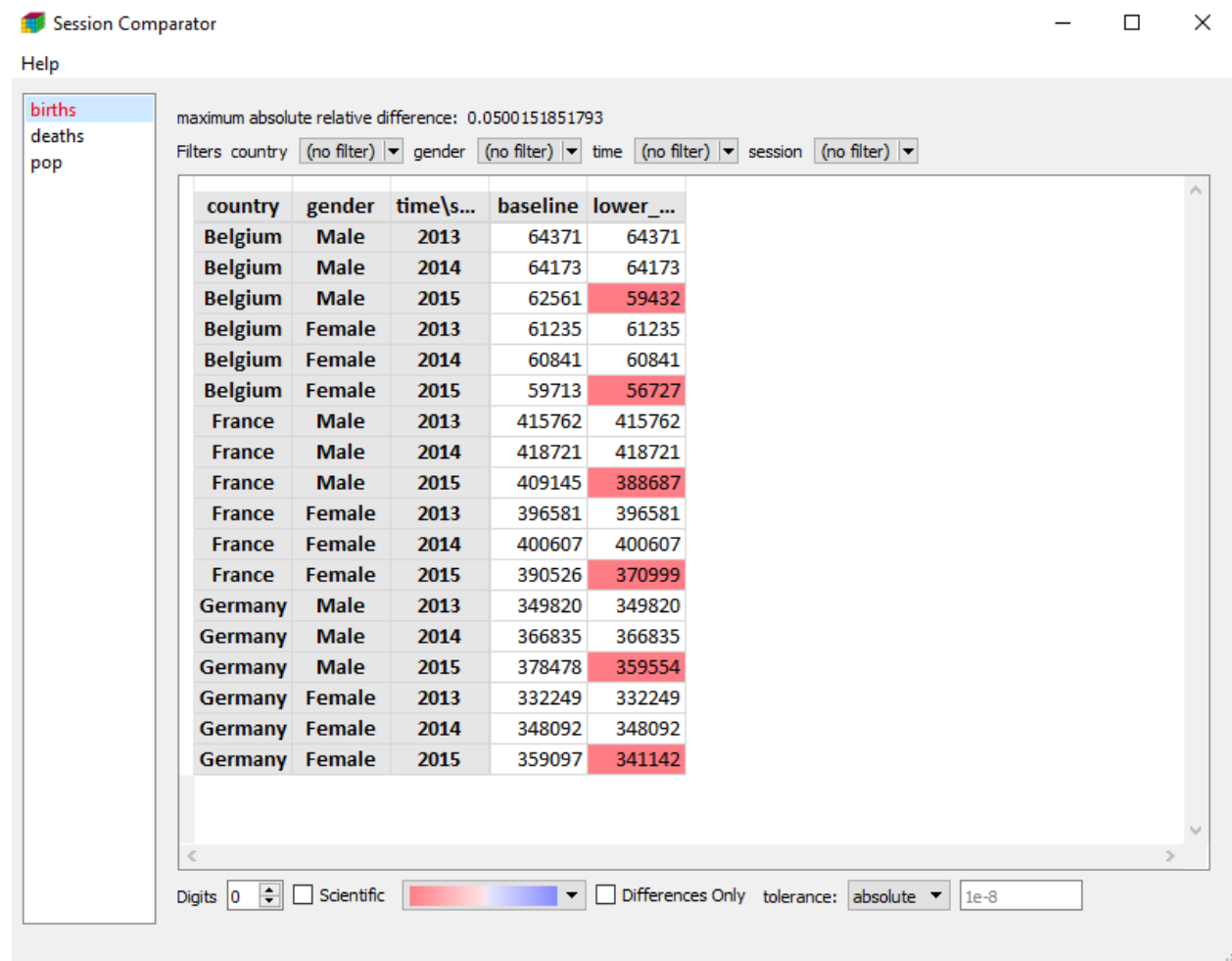
-----
NameError                                Traceback (most recent call last)
<ipython-input-30-18e0b26d24cb> in <module>
----> 1 s_check_different_values = s_pop != s_pop_copy
      2
      3 s_check_different_values.pop

NameError: name 's_pop' is not defined

```

A more visual way is to use the `compare` function which will open the Editor.

```
compare(s_pop, s_pop_alternative, names=['baseline', 'lower_birth_rate'])
```



Session API

Please go to the *Session* section of the API Reference to get the list of all methods of Session objects.

4.2.11 Compatibility with pandas

Import the LArray library:

```
[1]: from larray import *
```

To convert an Array object into a pandas DataFrame, the method *to_frame* can be used:

```
[2]: pop = load_example_data('demography_eurostat').pop
pop

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-2-cfd11435826> in <module>
----> 1 pop = load_example_data('demography_eurostat').pop
      2 pop

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/example.py in load_example_data(name)
      91     if name not in AVAILABLE_EXAMPLE_DATA.keys():
      92         raise ValueError("example_data must be chosen from list {}".
↳ format(list(AVAILABLE_EXAMPLE_DATA.keys())))
--> 93     return la.Session(AVAILABLE_EXAMPLE_DATA[name])

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in __init__(self, *args, **kwargs)
      94         if isinstance(a0, str):
      95             # assume a0 is a filename
--> 96             self.load(a0)
      97         else:
      98             # iterable of tuple or dict-like

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/core/session.py in load(self, fname, names, engine,
↳ display, **kwargs)
      426         else:
      427             handler = handler_cls(fname)
--> 428             metadata, objects = handler.read(names, display=display, **kwargs)
      429             for k, v in objects.items():
      430                 self[k] = v

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/common.py in read(self, keys, *args, **kwargs)
      128         print("loading", type, "object", key, "...", end=' ')
      129         try:
--> 130             res[key] = self._read_item(key, type, *args, **kwargs)
      131         except Exception:
      132             if not ignore_exceptions:

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in _read_item(self, key, type, *args,
↳ **kwargs)
      137         else:
```

(continues on next page)

(continued from previous page)

```

138         raise TypeError()
--> 139     return read_hdf(self.handle, hdf_key, *args, **kwargs)
140
141     def _dump_item(self, key, value, *args, **kwargs):

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/hdf.py in read_hdf(filepath_or_buffer, key, fill_
↳ value, na, sort_rows, sort_columns, name, **kwargs)
    81         cartesian_prod = writer != 'LArray'
    82         res = df_asarray(pd_obj, sort_rows=sort_rows, sort_columns=sort_
↳ columns, fill_value=fill_value,
--> 83             parse_header=False, cartesian_prod=cartesian_
↳ prod)
    84         if _meta is not None:
    85             res.meta = _meta

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in df_asarray(df, sort_rows, sort_
↳ columns, raw, parse_header, wide, cartesian_prod, **kwargs)
    338         unfold_last_axis_name = isinstance(axes_names[-1], basestring) and '\\'
↳ ' in axes_names[-1]
    339         res = from_frame(df, sort_rows=sort_rows, sort_columns=sort_columns,
↳ parse_header=parse_header,
--> 340             unfold_last_axis_name=unfold_last_axis_name,
↳ cartesian_prod=cartesian_prod, **kwargs)
    341
    342         # ugly hack to avoid anonymous axes converted as axes with name 'Unnamed:
↳ x' by pandas

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in from_frame(df, sort_rows, sort_
↳ columns, parse_header, unfold_last_axis_name, fill_value, meta, cartesian_prod,
↳ **kwargs)
    241         raise ValueError('sort_rows and sort_columns cannot not be used
↳ when cartesian_prod is set to False. '
    242             'Please call the method sort_axes on the
↳ returned array to sort rows or columns')
--> 243         axes_labels = index_to_labels(df.index, sort=False)
    244
    245         # Pandas treats column labels as column names (strings) so we need to
↳ convert them to values

~/checkouts/readthedocs.org/user_builds/larray/conda/0.32/lib/python3.6/site-packages/
↳ larray-0.32-py3.6.egg/larray/inout/pandas.py in index_to_labels(idx, sort)
    41         Returns unique labels for each dimension.
    42         """
--> 43         if isinstance(idx, pd.core.index.MultiIndex):
    44             if sort:
    45                 return list(idx.levels)

AttributeError: module 'pandas.core' has no attribute 'index'

```

```
[3]: df = pop.to_frame()
df
```

```
-----
NameError                                Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-3-002a3ec9888d> in <module>
----> 1 df = pop.to_frame()
      2 df
```

```
NameError: name 'pop' is not defined
```

Inversely, to convert a DataFrame into an Array object, use the function `asarray`:

```
[4]: pop = asarray(df)
      pop
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-5f4fb93339cc> in <module>
----> 1 pop = asarray(df)
      2 pop

NameError: name 'df' is not defined
```

4.3 API Reference

4.3.1 Axis

<code>Axis(labels[, name])</code>	Represents an axis.
-----------------------------------	---------------------

`larray.Axis`

class `larray.Axis` (*labels*, *name=None*)

Represents an axis. It consists of a name and a list of labels.

Parameters

labels [array-like or int] collection of values usable as labels, i.e. numbers or strings or the size of the axis. In the last case, a wildcard axis is created.

name [str or Axis, optional] name of the axis or another instance of Axis. In the second case, the name of the other axis is simply copied. By default None.

Examples

```
>>> gender = Axis(['M', 'F'], 'gender')
>>> gender
Axis(['M', 'F'], 'gender')
>>> gender.name
'gender'
>>> list(gender.labels)
['M', 'F']
```

using a string definition

```
>>> gender = Axis('gender=M,F')
>>> gender
Axis(['M', 'F'], 'gender')
>>> age = Axis('age=0..9')
>>> age
Axis([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 'age')
>>> code = Axis('code=A,C..E,F..G,Z')
>>> code
Axis(['A', 'C', 'D', 'E', 'F', 'G', 'Z'], 'code')
```

a wildcard axis only needs a length

```
>>> row = Axis(10, 'row')
>>> row
Axis(10, 'row')
>>> row.labels
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

axes can also be defined without name

```
>>> anonymous = Axis('0..4')
>>> anonymous
Axis([0, 1, 2, 3, 4], None)
```

Attributes

labels [array-like or int] labels of the axis.

name [str] name of the axis. None in the case of an anonymous axis.

__init__ (*self*, *labels*, *name=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self, labels[, name])</code>	Initialize self.
<code>align(self, other[, join])</code>	Align axis with other object using specified join method.
<code>all(self[, name])</code>	(Deprecated) Returns a group containing all labels.
<code>apply(self, func)</code>	Returns a new axis with the labels transformed by func.
<code>by(self, length[, step, template])</code>	Split axis into several groups of specified length.
<code>containing(self, substring)</code>	Returns a group with all the labels containing the specified substring.
<code>copy(self)</code>	Returns a copy of the axis.
<code>difference(self, other)</code>	Returns axis with the (set) difference of this axis labels and other labels.
<code>endingwith(self, suffix)</code>	Returns a group with the labels ending with the specified string.
<code>endswith(*args, **kwargs)</code>	
<code>equals(self, other)</code>	Checks if self is equal to another axis.
<code>extend(self, labels)</code>	Append new labels to an axis or increase its length in case of wildcard axis.

Continued on next page

Table 2 – continued from previous page

<code>group(self, *args, **kwargs)</code>	
<code>ignore_labels(self)</code>	Returns a wildcard axis with the same name and length than this axis.
<code>index(self, key)</code>	Translates a label key to its numerical index counterpart.
<code>insert(self, new_labels[, before, after])</code>	Return a new axis with <i>new_labels</i> inserted before <i>before</i> or after <i>after</i> .
<code>intersection(self, other)</code>	Returns axis with the (set) intersection of this axis labels and other labels.
<code>iscompatible(self, other)</code>	Checks if self is compatible with another axis.
<code>labels_summary(self)</code>	Returns a short representation of the labels.
<code>matches(*args, **kwargs)</code>	
<code>matching(self[, deprecated, pattern, regex])</code>	Returns a group with all the labels matching the specified pattern or regular expression.
<code>rename(self, name)</code>	Renames the axis.
<code>replace(self, old[, new])</code>	Returns a new axis with some labels replaced.
<code>split(self[, sep, names, regex, return_labels])</code>	Split axis and returns a list of Axis.
<code>startingwith(self, prefix)</code>	Returns a group with the labels starting with the specified string.
<code>startswith(*args, **kwargs)</code>	
<code>subaxis(self, key)</code>	Returns an axis for a sub-array.
<code>to_hdf(self, filepath[, key])</code>	Writes axis to a HDF file.
<code>translate(*args, **kwargs)</code>	
<code>union(self, other)</code>	Returns axis with the union of this axis labels and other labels.

Attributes

<code>dtype</code>	
<code>i</code>	Allows to define a subset using positions along the axis instead of labels.
<code>id</code>	
<code>iswildcard</code>	
<code>labels</code>	labels of the axis.
<code>name</code>	

Exploring

<code>Axis.name</code>	Name of the axis. None in the case of an anonymous axis.
<code>Axis.labels</code>	Labels of the axis.
<code>Axis.labels_summary</code>	Short representation of the labels.
<code>Axis.dtype</code>	Data type for the axis labels.

Copying

<code>Axis.copy(self)</code>	Returns a copy of the axis.
------------------------------	-----------------------------

larray.Axis.copy

`Axis.copy(self)`
Returns a copy of the axis.

Searching

<code>Axis.index(self, key)</code>	Translates a label key to its numerical index counterpart.
<code>Axis.containing(self, substring)</code>	Returns a group with all the labels containing the specified substring.
<code>Axis.startingwith(self, prefix)</code>	Returns a group with the labels starting with the specified string.
<code>Axis.endingwith(self, suffix)</code>	Returns a group with the labels ending with the specified string.
<code>Axis.matching(self[, deprecated, pattern, regex])</code>	Returns a group with all the labels matching the specified pattern or regular expression.

larray.Axis.index

`Axis.index(self, key)`
Translates a label key to its numerical index counterpart.

Parameters

key [key] Everything usable as a key.

Returns

(array of) int Numerical index(ices) of (all) label(s) represented by the key

Notes

Fancy index with boolean vectors are passed through unmodified

Examples

```
>>> people = Axis(['John Doe', 'Bruce Wayne', 'Bruce Willis', 'Waldo', 'Arthur_
↳Dent', 'Harvey Dent'], 'people')
>>> people.index('Waldo')
3
>>> people.index(people.containing('Bruce'))
array([1, 2])
```

larray.Axis.containing

`Axis.containing(self, substring)`

Returns a group with all the labels containing the specified substring.

Parameters

substring [str or Group] The substring to search for.

Returns

LGroup Group containing all the labels containing the substring.

Examples

```
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
>>> people.containing('Will')
people['Bruce Willis']
```

larray.Axis.startingwith

`Axis.startingwith(self, prefix)`

Returns a group with the labels starting with the specified string.

Parameters

prefix [str or Group] The prefix to search for.

Returns

LGroup Group containing all the labels starting with the given string.

Examples

```
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Waldo', 'Arthur Dent', 'Harvey_
↪Dent'], 'people')
>>> people.startingwith('Bru')
people['Bruce Wayne', 'Bruce Willis']
```

larray.Axis.endingwith

`Axis.endingwith(self, suffix)`

Returns a group with the labels ending with the specified string.

Parameters

suffix [str or Group] The suffix to search for.

Returns

LGroup Group containing all the labels ending with the given string.

Examples

```
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Waldo', 'Arthur Dent', 'Harvey_
↳Dent'], 'people')
>>> people.endswith('Dent')
people['Arthur Dent', 'Harvey Dent']
```

larray.Axis.matching

`Axis.matching` (*self*, *deprecated=None*, *pattern=None*, *regex=None*)

Returns a group with all the labels matching the specified pattern or regular expression.

Parameters

pattern [str or Group, optional] Pattern to match. * ? matches any single character * * matches any number of characters * [seq] matches any character in seq * [!seq] matches any character not in seq

To match any of the special characters above, wrap the character in brackets. For example, [?] matches the character ?.

regex [str or Group, optional] Regular expression pattern to match. Regular expressions are more powerful than what the simple patterns supported by the *pattern* argument but are also more complex to write. See [Regular Expression](#) for more details about how to build a regular expression pattern.

Returns

LGroup Group containing all the labels matching the pattern.

Examples

```
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Waldo', 'Arthur Dent', 'Harvey_
↳Dent'], 'people')
```

```
>>> # All labels starting with "A" and ending with "t"
>>> people.matching(pattern='A*t')
people['Arthur Dent']
>>> # All labels containing "W" and ending with "s"
>>> people.matching(pattern='*W*s')
people['Bruce Willis']
>>> # All labels with exactly 5 characters
>>> people.matching(pattern='?????')
people['Waldo']
>>> # All labels starting with either "A" or "B"
>>> people.matching(pattern='[AB]*')
people['Bruce Wayne', 'Bruce Willis', 'Arthur Dent']
```

Regular expressions are more powerful but usually harder to write and less readable

```
>>> # All labels starting with "W" and ending with "o"
>>> people.matching(regex='A.*t')
people['Arthur Dent']
>>> # All labels not containing character "a"
>>> people.matching(regex='^[^a]*$')
people['Bruce Willis', 'Arthur Dent']
```


Modifying/Selecting

<code>Axis.__getitem__(self, key)</code>	Returns a group (list or unique element) of label(s) usable in <code>.sum</code> or <code>.filter</code>
<code>Axis.i</code>	Allows to define a subset using positions along the axis instead of labels.
<code>Axis.by(self, length[, step, template])</code>	Split axis into several groups of specified length.
<code>Axis.rename(self, name)</code>	Renames the axis.
<code>Axis.subaxis(self, key)</code>	Returns an axis for a sub-array.
<code>Axis.extend(self, labels)</code>	Append new labels to an axis or increase its length in case of wildcard axis.
<code>Axis.insert(self, new_labels[, before, after])</code>	Return a new axis with <i>new_labels</i> inserted before <i>before</i> or after <i>after</i> .
<code>Axis.replace(self, old[, new])</code>	Returns a new axis with some labels replaced.
<code>Axis.apply(self, func)</code>	Returns a new axis with the labels transformed by <i>func</i> .
<code>Axis.union(self, other)</code>	Returns axis with the union of this axis labels and other labels.
<code>Axis.intersection(self, other)</code>	Returns axis with the (set) intersection of this axis labels and other labels.
<code>Axis.difference(self, other)</code>	Returns axis with the (set) difference of this axis labels and other labels.
<code>Axis.align(self, other[, join])</code>	Align axis with other object using specified join method.
<code>Axis.split(self[, sep, names, regex, ...])</code>	Split axis and returns a list of Axis.
<code>Axis.ignore_labels(self)</code>	Returns a wildcard axis with the same name and length than this axis.

larray.Axis.__getitem__

`Axis.__getitem__(self, key)`

Returns a group (list or unique element) of label(s) usable in `.sum` or `.filter`

key is a label-based key (other axis, slice and fancy indexing are supported)

Returns

Group group containing selected label(s)/position(s).

Notes

key is label-based (slice and fancy indexing are supported)

larray.Axis.i

`Axis.i`

Allows to define a subset using positions along the axis instead of labels.

Examples

```
>>> from larray import ndtest
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> arr = ndtest([sex, time])
>>> arr
sex\\time  2007  2008  2009  2010
      M      0      1      2      3
      F      4      5      6      7
>>> arr[time.i[0, -1]]
sex\\time  2007  2010
      M      0      3
      F      4      7
```

larray.Axis.by

Axis **.by** (*self*, *length*, *step=None*, *template=None*)
Split axis into several groups of specified length.

Parameters

length [int] length of groups

step [int, optional] step between groups. Defaults to length.

template [str, optional] template describing how group names are generated. It is a string containing specific arguments written inside brackets {}. Available arguments are {start} and {end} representing the first and last label of each group. By default, template is defined as '{start}:{end}'.

Returns

list of Group

Notes

step can be smaller than length, in which case, this will produce overlapping groups.

Examples

```
>>> age = Axis('age=0..6')
>>> age
Axis([0, 1, 2, 3, 4, 5, 6], 'age')
>>> age.by(3)
(age.i[0:3] >> '0:2', age.i[3:6] >> '3:5', age.i[6:7] >> '6')
>>> age.by(3, step=2)
(age.i[0:3] >> '0:2', age.i[2:5] >> '2:4', age.i[4:7] >> '4:6', age.i[6:7] >> '6')
>>> age.by(3, template='{start}-{end}')
(age.i[0:3] >> '0-2', age.i[3:6] >> '3-5', age.i[6:7] >> '6')
```

larray.Axis.rename

`Axis.rename(self, name)`

Renames the axis.

Parameters

name [str] the new name for the axis.

Returns

Axis a new Axis with the same labels but a different name.

Examples

```

>>> sex = Axis('sex=M,F')
>>> sex
Axis(['M', 'F'], 'sex')
>>> sex.rename('gender')
Axis(['M', 'F'], 'gender')

```

larray.Axis.subaxis

`Axis.subaxis(self, key)`

Returns an axis for a sub-array.

Parameters

key [int, or collection (list, slice, array, Array) of them] Indices-based key to use for the new axis.

Returns

Axis Subaxis. If key is a None slice, the original Axis is returned. If key is an Array, the list of axes is returned.

Examples

```

>>> age = Axis(range(100), 'age')
>>> age.subaxis(range(10, 19))
Axis([10, 11, 12, 13, 14, 15, 16, 17, 18], 'age')

```

larray.Axis.extend

`Axis.extend(self, labels)`

Append new labels to an axis or increase its length in case of wildcard axis. Note that *extend* does not occur in-place: a new axis object is allocated, filled and returned.

Parameters

labels [int, iterable or Axis] New labels to append to the axis. Passing directly another Axis is also possible. If the current axis is a wildcard axis, passing a length is enough.

Returns

Axis A copy of the axis with new labels appended to it or with increased length (if wildcard).

Examples

```
>>> time = Axis([2007, 2008], 'time')
>>> time
Axis([2007, 2008], 'time')
>>> time.extend([2009, 2010])
Axis([2007, 2008, 2009, 2010], 'time')
>>> waxis = Axis(10, 'wildcard_axis')
>>> waxis
Axis(10, 'wildcard_axis')
>>> waxis.extend(5)
Axis(15, 'wildcard_axis')
>>> waxis.extend([11, 12, 13, 14])
Traceback (most recent call last):
...
ValueError: Axis to append must (not) be wildcard if self is (not) wildcard
```

larray.Axis.insert

Axis.insert (*self*, *new_labels*, *before=None*, *after=None*)

Return a new axis with *new_labels* inserted before *before* or after *after*.

Parameters

new_labels [scalar, tuple/list/array of scalars, Group or Axis] New label(s) to append to the axis.

before [scalar or Group, optional] Label or group before which to insert *new_labels*.

after [scalar or Group, optional] Label or group after which to insert *new_labels*.

Returns

Axis A copy of the axis with the new labels inserted.

Examples

```
>>> time = Axis([2007, 2009], 'time')
>>> time.insert(2008, before=2009)
Axis([2007, 2008, 2009], 'time')
>>> time.insert(2008, after=2007)
Axis([2007, 2008, 2009], 'time')
>>> time.insert(2008, before=time.i[1])
Axis([2007, 2008, 2009], 'time')
>>> time.insert(2008, after=time.i[0])
Axis([2007, 2008, 2009], 'time')
>>> b = Axis(['b1', 'b2'], 'b')
>>> b.insert('b1.5', before='b2')
Axis(['b1', 'b1.5', 'b2'], 'b')
>>> b.insert(['b1.1', 'b1.2'], before='b2')
Axis(['b1', 'b1.1', 'b1.2', 'b2'], 'b')
>>> c = Axis(['c1', 'c2'], 'c')
>>> b.insert(c, before='b2')
Axis(['b1', 'c1', 'c2', 'b2'], 'b')
```

larray.Axis.replace

`Axis.replace(self, old, new=None)`

Returns a new axis with some labels replaced.

Parameters

old [any scalar (bool, int, str, ...), tuple/list/array of scalars, or a mapping.] the label(s) to be replaced. Old can be a mapping {old1: new1, old2: new2, ...}

new [any scalar (bool, int, str, ...) or tuple/list/array of scalars, optional] the new label(s). This is argument must not be used if old is a mapping.

Returns

Axis a new Axis with the old labels replaced by new labels.

Examples

```

>>> sex = Axis('sex=M,F')
>>> sex
Axis(['M', 'F'], 'sex')
>>> sex.replace('M', 'Male')
Axis(['Male', 'F'], 'sex')
>>> sex.replace({'M': 'Male', 'F': 'Female'})
Axis(['Male', 'Female'], 'sex')
>>> sex.replace(['M', 'F'], ['Male', 'Female'])
Axis(['Male', 'Female'], 'sex')

```

larray.Axis.apply

`Axis.apply(self, func)`

Returns a new axis with the labels transformed by func.

Parameters

func [callable] A callable which takes a single argument and returns a single value.

Returns

Axis a new Axis with the transformed labels.

Examples

```

>>> sex = Axis('sex=MALE,FEMALE')
>>> sex.apply(str.capitalize)
Axis(['Male', 'Female'], 'sex')

```

larray.Axis.union

`Axis.union(self, other)`

Returns axis with the union of this axis labels and other labels.

Labels relative order will be kept intact, but only unique labels will be returned. Labels from this axis will be before labels from other.

Parameters

other [Axis or any sequence of labels] other labels

Returns

Axis

Examples

```
>>> a = Axis('a=a0..a2')
>>> a.union('a1')
Axis(['a0', 'a1', 'a2'], 'a')
>>> a.union('a3')
Axis(['a0', 'a1', 'a2', 'a3'], 'a')
>>> a.union(Axis('a=a1..a3'))
Axis(['a0', 'a1', 'a2', 'a3'], 'a')
>>> a.union('a1..a3')
Axis(['a0', 'a1', 'a2', 'a3'], 'a')
>>> a.union(['a1', 'a2', 'a3'])
Axis(['a0', 'a1', 'a2', 'a3'], 'a')
```

larray.Axis.intersection

`Axis.intersection(self, other)`

Returns axis with the (set) intersection of this axis labels and other labels.

In other words, this will use labels from this axis if they are also in other. Labels relative order will be kept intact.

Parameters

other [Axis or any sequence of labels] other labels

Returns

Axis

Examples

```
>>> a = Axis('a=a0..a2')
>>> a.intersection('a1')
Axis(['a1'], 'a')
>>> a.intersection('a3')
Axis([], 'a')
>>> a.intersection(Axis('a=a1..a3'))
Axis(['a1', 'a2'], 'a')
>>> a.intersection('a1..a3')
```

(continues on next page)

(continued from previous page)

```
Axis(['a1', 'a2'], 'a')
>>> a.intersection(['a1', 'a2', 'a3'])
Axis(['a1', 'a2'], 'a')
```

larray.Axis.difference**Axis.difference** (*self*, *other*)

Returns axis with the (set) difference of this axis labels and other labels.

In other words, this will use labels from this axis if they are not in other. Labels relative order will be kept intact.

Parameters**other** [Axis or any sequence of labels] other labels**Returns****Axis****Examples**

```
>>> a = Axis('a=a0..a2')
>>> a.difference('a1')
Axis(['a0', 'a2'], 'a')
>>> a.difference('a3')
Axis(['a0', 'a1', 'a2'], 'a')
>>> a.difference(Axis('a=a1..a3'))
Axis(['a0'], 'a')
>>> a.difference('a1..a3')
Axis(['a0'], 'a')
>>> a.difference(['a1', 'a2', 'a3'])
Axis(['a0'], 'a')
```

larray.Axis.align**Axis.align** (*self*, *other*, *join*='outer')

Align axis with other object using specified join method.

Parameters**other** [Axis or label sequence]**join** [{ 'outer', 'inner', 'left', 'right', 'exact' }, optional] Defaults to 'outer'.**Returns****Axis** Aligned axis**See also:***Array.align*

Examples

```
>>> axis1 = Axis('a=a0..a2')
>>> axis2 = Axis('a=a1..a3')
>>> axis1.align(axis2)
Axis(['a0', 'a1', 'a2', 'a3'], 'a')
>>> axis1.align(axis2, join='inner')
Axis(['a1', 'a2'], 'a')
>>> axis1.align(axis2, join='left')
Axis(['a0', 'a1', 'a2'], 'a')
>>> axis1.align(axis2, join='right')
Axis(['a1', 'a2', 'a3'], 'a')
>>> axis1.align(axis2, join='exact')
Traceback (most recent call last):
...
ValueError: align method with join='exact' expected
Axis(['a0', 'a1', 'a2'], 'a') to be equal to Axis(['a1', 'a2', 'a3'], 'a')
```

larray.Axis.split

`Axis.split` (*self*, *sep*='_', *names*=None, *regex*=None, *return_labels*=False)

Split axis and returns a list of Axis.

Parameters

sep [str, optional] Delimiter to use for splitting. Defaults to '_'. When *regex* is provided, the delimiter is only used on *names* if given as one string or on axis name if *names* is None.

names [str or list of str, optional] Names of resulting axes. Defaults to None.

regex [str, optional] Use regex instead of delimiter to split labels. Defaults to None.

labels [bool, optional] Whether or not split labels must be returned (as a tuple of tuples). These labels are suitable for indexing via `array.points[labels]`. Defaults to False.

Returns

list of Axis or (list of Axis, array-like)

Examples

```
>>> a_b = Axis('a_b=a0_b0,a0_b1,a0_b2,a1_b0,a1_b1,a1_b2')
>>> a_b.split()
[Axis(['a0', 'a1'], 'a'), Axis(['b0', 'b1', 'b2'], 'b')]
```

larray.Axis.ignore_labels

`Axis.ignore_labels` (*self*)

Returns a wildcard axis with the same name and length than this axis.

Useful when you want to apply operations between two arrays with the same shape but incompatible axes (different labels).

Returns

Axis

Examples

```
>>> a = Axis('a=a1,a2')
>>> a
Axis(['a1', 'a2'], 'a')
>>> a.ignore_labels()
Axis(2, 'a')
```

Testing

<code>Axis.iscompatible(self, other)</code>	Checks if self is compatible with another axis.
<code>Axis.equals(self, other)</code>	Checks if self is equal to another axis.

larray.Axis.iscompatible

`Axis.iscompatible(self, other)`

Checks if self is compatible with another axis.

- Two non-wildcard axes are compatible if they have the same name and labels.
- A wildcard axis of length 1 is compatible with any other axis sharing the same name.
- A wildcard axis of length > 1 is compatible with any axis of the same length or length 1 and sharing the same name.

Parameters

other [Axis] Axis to compare with.

Returns

bool True if input axis is compatible with self, False otherwise.

Examples

```
>>> a10 = Axis(range(10), 'a')
>>> wa10 = Axis(10, 'a')
>>> wa1 = Axis(1, 'a')
>>> b10 = Axis(range(10), 'b')
>>> a10.iscompatible(b10)
False
>>> a10.iscompatible(wa10)
True
>>> a10.iscompatible(wa1)
True
>>> wa1.iscompatible(b10)
False
```

larray.Axis.equals

`Axis.equals(self, other)`

Checks if self is equal to another axis. Two axes are equal if they have the same name and label(s).

Parameters

other [Axis] Axis to compare with.

Returns

bool True if input axis is equal to self, False otherwise.

Examples

```
>>> age = Axis(range(5), 'age')
>>> age_2 = Axis(5, 'age')
>>> age_3 = Axis(range(5), 'young children')
>>> age_4 = Axis([0, 1, 2, 3, 4], 'age')
>>> age.equals(age_2)
False
>>> age.equals(age_3)
False
>>> age.equals(age_4)
True
```

Save

`Axis.to_hdf(self, filepath[, key])`

Writes axis to a HDF file.

larray.Axis.to_hdf

`Axis.to_hdf(self, filepath, key=None)`

Writes axis to a HDF file.

A HDF file can contain multiple axes. The ‘key’ parameter is a unique identifier for the axis.

Parameters

filepath [str] Path where the hdf file has to be written.

key [str or Group, optional] Key (path) of the axis within the HDF file (see Notes below). If None, the name of the axis is used. Defaults to None.

Notes

Objects stored in a HDF file can be grouped together in *HDF groups*. If an object ‘my_obj’ is stored in a HDF group ‘my_group’, the key associated with this object is then ‘my_group/my_obj’. Be aware that a HDF group can have subgroups.

Examples

```
>>> a = Axis("a=a0..a2")
```

Save axis

```
>>> # by default, the key is the name of the axis
>>> a.to_hdf('test.h5')
```

Save axis with a specific key

```
>>> a.to_hdf('test.h5', 'a')
```

Save axis in a specific HDF group

```
>>> a.to_hdf('test.h5', 'axes/a')
```

4.3.2 Group

IGroup

IGroup(key[, name, axis])

Index Group.

larray.IGroup

class larray.IGroup (key, name=None, axis=None)

Index Group.

Represents a subset of indices of an axis.

Parameters

key [key] Anything usable for indexing. A key should be either a single position, a sequence of positions, or a slice with integer bounds.

name [str, optional] Name of the group.

axis [int, str, Axis, optional] Axis for group.

__init__ (self, key, name=None, axis=None)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self, key[, name, axis])</code>	Initialize self.
<code>by(self, length[, step, template])</code>	Split group into several groups of specified length.
<code>containing(self, substring)</code>	Returns a group with all the labels containing the specified substring.
<code>difference(self, other)</code>	Returns (set) difference of this label group and other.
<code>endingwith(self, suffix)</code>	Returns a group with the labels ending with the specified string.
<code>equals(self, other)</code>	Checks if this group is equal to another group.
<code>eval(self)</code>	Translate key to labels, if it is not already, expanding slices in the process.
<code>intersection(self, other)</code>	Returns (set) intersection of this label group and other.
<code>matching(self[, deprecated, pattern, regex])</code>	Returns a group with all the labels matching the specified pattern or regular expression.
<code>named(self, name)</code>	Returns group with a different name.
<code>retarget_to(self, target_axis)</code>	Retarget group to another axis.
<code>set(self)</code>	Creates LSet from this group
<code>startingwith(self, prefix)</code>	Returns a group with the labels starting with the specified string.
<code>to_hdf(self, filepath[, key, axis_key])</code>	Writes group to a HDF file.
<code>to_label(self)</code>	Translate key to labels, if it is not already
<code>translate(self[, bound, stop])</code>	compute position(s) of group
<code>union(self, other)</code>	Returns (set) union of this label group and other.
<code>with_axis(self, axis)</code>	Returns group with a different axis.

Attributes

<code>axis</code>
<code>format_string</code>
<code>key</code>
<code>name</code>

<code>IGroup.named(self, name)</code>	Returns group with a different name.
<code>IGroup.with_axis(self, axis)</code>	Returns group with a different axis.
<code>IGroup.by(self, length[, step, template])</code>	Split group into several groups of specified length.
<code>IGroup.equals(self, other)</code>	Checks if this group is equal to another group.
<code>IGroup.translate(self[, bound, stop])</code>	compute position(s) of group
<code>IGroup.union(self, other)</code>	Returns (set) union of this label group and other.
<code>IGroup.intersection(self, other)</code>	Returns (set) intersection of this label group and other.
<code>IGroup.difference(self, other)</code>	Returns (set) difference of this label group and other.
<code>IGroup.containing(self, substring)</code>	Returns a group with all the labels containing the specified substring.
<code>IGroup.startingwith(self, prefix)</code>	Returns a group with the labels starting with the specified string.
<code>IGroup.endingwith(self, suffix)</code>	Returns a group with the labels ending with the specified string.

Continued on next page

Table 12 – continued from previous page

<code>IGroup.matching(self[, deprecated, pattern, ...])</code>	Returns a group with all the labels matching the specified pattern or regular expression.
<code>IGroup.to_hdf(self, filepath[, key, axis_key])</code>	Writes group to a HDF file.

larray.IGroup.named`IGroup.named(self, name)`

Returns group with a different name.

Parameters**name** [str] new name for group**Returns****Group****larray.IGroup.with_axis**`IGroup.with_axis(self, axis)`

Returns group with a different axis.

Parameters**axis** [int, str, Axis] new axis for group**Returns****Group****larray.IGroup.by**`IGroup.by(self, length, step=None, template=None)`

Split group into several groups of specified length.

Parameters**length** [int] length of new groups**step** [int, optional] step between groups. Defaults to length.**template** [str, optional] template describing how group names are generated. It is a string containing specific arguments written inside brackets {}. Available arguments are {start} and {end} representing the first and last label of each group. By default, template is defined as '{start}:{end}'.**Returns****list of Group**

Notes

step can be smaller than length, in which case, this will produce overlapping groups.

Examples

```
>>> from larray import Axis, X
>>> age = Axis('age=0..100')
>>> young_children = age[0:6]
>>> young_children.by(3)
(age.i[0:3] >> '0:2', age.i[3:6] >> '3:5', age.i[6:7] >> '6')
>>> young_children.by(3, step=2)
(age.i[0:3] >> '0:2', age.i[2:5] >> '2:4', age.i[4:7] >> '4:6', age.i[6:7] >> '6')
>>> young_children.by(3, template='{start}-{end}')
(age.i[0:3] >> '0-2', age.i[3:6] >> '3-5', age.i[6:7] >> '6')
```

larray.IGroup.equals

IGroup.**equals** (*self*, *other*)

Checks if this group is equal to another group. Two groups are equal if they have the same group and axis names and correspond to the same labels.

Parameters

other [Group] Group to compare with.

Returns

bool True if the other group is equal to this group, False otherwise.

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a3')
>>> a02 = a['a0:a2'] >> 'group_a'
```

Same group names, axis names and labels

```
>>> a02.equals(a02)
True
```

Different group names (one is None)

```
>>> a02.equals(a['a0:a2'])
False
```

Different axis name

```
>>> other_axis = a.rename('other_name')
>>> a02.equals(other_axis['a0:a2'] >> 'group_a')
False
```

Different labels

```
>>> a02.equals(a['a1:a3'] >> 'group_a')
False
```

Mixing slice and list groups

```
>>> a['a0:a2'].equals(a['a0,a1,a2'])
True
```

Mixing LGroup and IGroup

```
>>> a['a0:a2'].equals(a.i[0:3])
True
```

larray.IGroup.translate

IGroup.translate (*self*, *bound=None*, *stop=False*)
compute position(s) of group

larray.IGroup.union

IGroup.union (*self*, *other*)
Returns (set) union of this label group and other.

Labels relative order will be kept intact, but only unique labels will be returned. Labels from this group will be before labels from other.

Parameters

other [Group or any sequence of labels] other labels

Returns

LSet

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a2')
>>> a['a0', 'a1'].union(a['a1', 'a2'])
a['a0', 'a1', 'a2'].set()
>>> a['a0', 'a1'].union('a1,a2')
a['a0', 'a1', 'a2'].set()
>>> a['a0', 'a1'].union(['a1', 'a2'])
a['a0', 'a1', 'a2'].set()
```

larray.IGroup.intersection

IGroup.**intersection** (*self*, *other*)

Returns (set) intersection of this label group and other.

In other words, this will return labels from this group which are also in other. Labels relative order will be kept intact, but only unique labels will be returned.

Parameters

other [Group or any sequence of labels] other labels

Returns

LSet

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a2')
>>> a['a0', 'a1'].intersection(a['a1', 'a2'])
a['a1'].set()
>>> a['a0', 'a1'].intersection('a1,a2')
a['a1'].set()
>>> a['a0', 'a1'].intersection(['a1', 'a2'])
a['a1'].set()
```

larray.IGroup.difference

IGroup.**difference** (*self*, *other*)

Returns (set) difference of this label group and other.

In other words, this will return labels from this group without those in other. Labels relative order will be kept intact, but only unique labels will be returned.

Parameters

other [Group or any sequence of labels] other labels

Returns

LSet

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a2')
>>> a['a0', 'a1'].difference(a['a1', 'a2'])
a['a0'].set()
>>> a['a0', 'a1'].difference('a1,a2')
a['a0'].set()
>>> a['a0', 'a1'].difference(['a1', 'a2'])
a['a0'].set()
```


`larray.LGroup.containing`

`IGroup.containing` (*self*, *substring*)

Returns a group with all the labels containing the specified substring.

Parameters

substring [str or Group] The substring to search for.

Returns

LGroup Group containing all the labels containing the substring.

Examples

```

>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
>>> group = people.startingwith('Bru')
>>> group
people['Bruce Wayne', 'Bruce Willis']
>>> group.containing('Will')
people['Bruce Willis']

```

`larray.LGroup.startingwith`

`IGroup.startingwith` (*self*, *prefix*)

Returns a group with the labels starting with the specified string.

Parameters

prefix [str or Group] The prefix to search for.

Returns

LGroup Group containing all the labels starting with the given string.

Examples

```

>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Arthur Dent', 'Harvey Dent'], 'people')
>>> group = people.endingwith('Dent')
>>> group
people['Arthur Dent', 'Harvey Dent']
>>> group.startingwith('Art')
people['Arthur Dent']

```

larray.IGroup.endingwith

`IGroup.endingwith(self, suffix)`

Returns a group with the labels ending with the specified string.

Parameters

suffix [str or Group] The suffix to search for.

Returns

LGroup Group containing all the labels ending with the given string.

Examples

```
>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
>>> group = people.startingwith('Bru')
>>> group
people['Bruce Wayne', 'Bruce Willis']
>>> people.endingwith('yne')
people['Bruce Wayne']
```

larray.IGroup.matching

`IGroup.matching(self, deprecated=None, pattern=None, regex=None)`

Returns a group with all the labels matching the specified pattern or regular expression.

Parameters

pattern [str or Group] Pattern to match.

- `?` matches any single character
- `*` matches any number of characters
- `[seq]` matches any character in seq
- `[!seq]` matches any character not in seq

To match any of the special characters above, wrap the character in brackets. For example, `[?]` matches the character `?`.

regex [str or Group] Regular expression pattern to match. Regular expressions are more powerful than what the simple patterns supported by the *pattern* argument but are also more complex to write. See [Regular Expression](#) for more details about how to build a regular expression pattern.

Returns

LGroup Group containing all the labels matching the pattern.

Examples

```
>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
```

Let us create a group with all names starting with B

```
>>> group = people.startingwith('B')
>>> group
people['Bruce Wayne', 'Bruce Willis']
```

Within that group, all labels containing any characters then W then any characters then s are given by

```
>>> group.matching(pattern='*W*s')
people['Bruce Willis']
```

Regular expressions are more powerful but usually harder to write and less readable. For example, here are the labels not containing the letter “i”.

```
>>> group.matching(regex='^[^i]*$')
people['Bruce Wayne']
```

larray.IGroup.to_hdf

IGroup.**to_hdf** (*self*, *filepath*, *key=None*, *axis_key=None*)

Writes group to a HDF file.

A HDF file can contain multiple groups. The ‘key’ parameter is a unique identifier for the group. The ‘axis_key’ parameter is the unique identifier for the associated axis. The associated axis will be saved if not already present in the HDF file.

Parameters

filepath [str] Path where the hdf file has to be written.

key [str or Group, optional] Key (path) of the group within the HDF file (see Notes below). If None, the name of the group is used. Defaults to None.

axis_key [str, optional] Key (path) of the associated axis in the HDF file (see Notes below). If None, the name of the axis associated with the group is used. Defaults to None.

Notes

Objects stored in a HDF file can be grouped together in *HDF groups*. If an object ‘my_obj’ is stored in a HDF group ‘my_group’, the key associated with this object is then ‘my_group/my_obj’. Be aware that a HDF group can have subgroups.

Examples

```
>>> from larray import Axis
>>> a = Axis("a=a0..a2")
>>> a.to_hdf('test.h5')
>>> a01 = a['a0,a1'] >> 'a01'
```

Save group

```
>>> # by default, the key is the name of the group
>>> # and axis_key the name of the associated axis
>>> a01.to_hdf('test.h5')
```

Save group with a specific key

```
>>> a01.to_hdf('test.h5', 'a_01')
```

Save group in a specific HDF group

```
>>> a.to_hdf('test.h5', 'groups/a01')
```

The associated axis is saved with the group if not already present in the HDF file

```
>>> b = Axis("b=b0..b2")
>>> b01 = b['b0,b1'] >> 'b01'
>>> # save both the group 'b01' and the associated axis 'b'
>>> b01.to_hdf('test.h5')
```

LGroup

LGroup(key[, name, axis])

Label group.

larray.LGroup

class larray.LGroup (key, name=None, axis=None)

Label group.

Represents a subset of labels of an axis.

Parameters

key [key] Anything usable for indexing. A key should be either sequence of labels, a slice with label bounds or a string.

name [str, optional] Name of the group.

axis [int, str, Axis, optional] Axis for group.

Examples

```
>>> from larray import Axis, X
>>> age = Axis('0..100', 'age')
>>> teens = X.age[10:19].named('teens')
>>> teens
X.age[10:19] >> 'teens'
>>> teens = X.age[10:19] >> 'teens'
>>> teens
X.age[10:19] >> 'teens'
```

__init__(*self*, *key*, *name=None*, *axis=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self, key[, name, axis])</code>	Initialize self.
<code>by(self, length[, step, template])</code>	Split group into several groups of specified length.
<code>containing(self, substring)</code>	Returns a group with all the labels containing the specified substring.
<code>difference(self, other)</code>	Returns (set) difference of this label group and other.
<code>endingwith(self, suffix)</code>	Returns a group with the labels ending with the specified string.
<code>equals(self, other)</code>	Checks if this group is equal to another group.
<code>eval(self)</code>	Translate key to labels, if it is not already, expanding slices in the process.
<code>intersection(self, other)</code>	Returns (set) intersection of this label group and other.
<code>matching(self[, deprecated, pattern, regex])</code>	Returns a group with all the labels matching the specified pattern or regular expression.
<code>named(self, name)</code>	Returns group with a different name.
<code>retarget_to(self, target_axis)</code>	Retarget group to another axis.
<code>set(self)</code>	Creates LSet from this group
<code>startingwith(self, prefix)</code>	Returns a group with the labels starting with the specified string.
<code>to_hdf(self, filepath[, key, axis_key])</code>	Writes group to a HDF file.
<code>to_label(self)</code>	Translate key to labels, if it is not already
<code>translate(self[, bound, stop])</code>	compute position(s) of group
<code>union(self, other)</code>	Returns (set) union of this label group and other.
<code>with_axis(self, axis)</code>	Returns group with a different axis.

Attributes

<code>axis</code>
<code>format_string</code>
<code>key</code>
<code>name</code>

<i>LGroup.named</i> (self, name)	Returns group with a different name.
<i>LGroup.with_axis</i> (self, axis)	Returns group with a different axis.
<i>LGroup.by</i> (self, length[, step, template])	Split group into several groups of specified length.
<i>LGroup.equals</i> (self, other)	Checks if this group is equal to another group.
<i>LGroup.translate</i> (self[, bound, stop])	compute position(s) of group
<i>LGroup.union</i> (self, other)	Returns (set) union of this label group and other.
<i>LGroup.intersection</i> (self, other)	Returns (set) intersection of this label group and other.
<i>LGroup.difference</i> (self, other)	Returns (set) difference of this label group and other.
<i>LGroup.containing</i> (self, substring)	Returns a group with all the labels containing the specified substring.
<i>LGroup.startingwith</i> (self, prefix)	Returns a group with the labels starting with the specified string.
<i>LGroup.endingwith</i> (self, suffix)	Returns a group with the labels ending with the specified string.
<i>LGroup.matching</i> (self[, deprecated, pattern, ...])	Returns a group with all the labels matching the specified pattern or regular expression.
<i>LGroup.to_hdf</i> (self, filepath[, key, axis_key])	Writes group to a HDF file.

larray.LGroup.named

`LGroup.named` (*self*, *name*)

Returns group with a different name.

Parameters

name [str] new name for group

Returns

Group

larray.LGroup.with_axis

`LGroup.with_axis` (*self*, *axis*)

Returns group with a different axis.

Parameters

axis [int, str, Axis] new axis for group

Returns

Group

`larray.LGroup.by`

`LGroup.by` (*self*, *length*, *step=None*, *template=None*)
 Split group into several groups of specified length.

Parameters

length [int] length of new groups

step [int, optional] step between groups. Defaults to length.

template [str, optional] template describing how group names are generated. It is a string containing specific arguments written inside brackets {}. Available arguments are {start} and {end} representing the first and last label of each group. By default, template is defined as '{start}:{end}'.

Returns

list of Group

Notes

step can be smaller than length, in which case, this will produce overlapping groups.

Examples

```
>>> from larray import Axis, X
>>> age = Axis('age=0..100')
>>> young_children = age[0:6]
>>> young_children.by(3)
(age.i[0:3] >> '0:2', age.i[3:6] >> '3:5', age.i[6:7] >> '6')
>>> young_children.by(3, step=2)
(age.i[0:3] >> '0:2', age.i[2:5] >> '2:4', age.i[4:7] >> '4:6', age.i[6:7] >> '6')
>>> young_children.by(3, template='{start}-{end}')
(age.i[0:3] >> '0-2', age.i[3:6] >> '3-5', age.i[6:7] >> '6')
```

`larray.LGroup.equals`

`LGroup.equals` (*self*, *other*)

Checks if this group is equal to another group. Two groups are equal if they have the same group and axis names and correspond to the same labels.

Parameters

other [Group] Group to compare with.

Returns

bool True if the other group is equal to this group, False otherwise.

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a3')
>>> a02 = a['a0:a2'] >> 'group_a'
```

Same group names, axis names and labels

```
>>> a02.equals(a02)
True
```

Different group names (one is None)

```
>>> a02.equals(a['a0:a2'])
False
```

Different axis name

```
>>> other_axis = a.rename('other_name')
>>> a02.equals(other_axis['a0:a2'] >> 'group_a')
False
```

Different labels

```
>>> a02.equals(a['a1:a3'] >> 'group_a')
False
```

Mixing slice and list groups

```
>>> a['a0:a2'].equals(a['a0,a1,a2'])
True
```

Mixing LGroup and IGroup

```
>>> a['a0:a2'].equals(a.i[0:3])
True
```

larray.LGroup.translate

LGroup.**translate** (*self*, *bound=None*, *stop=False*)
compute position(s) of group

larray.LGroup.union

LGroup.**union** (*self*, *other*)

Returns (set) union of this label group and other.

Labels relative order will be kept intact, but only unique labels will be returned. Labels from this group will be before labels from other.

Parameters

other [Group or any sequence of labels] other labels

Returns

LSet

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a2')
>>> a['a0', 'a1'].union(a['a1', 'a2'])
a['a0', 'a1', 'a2'].set()
>>> a['a0', 'a1'].union('a1,a2')
a['a0', 'a1', 'a2'].set()
>>> a['a0', 'a1'].union(['a1', 'a2'])
a['a0', 'a1', 'a2'].set()
```

larray.LGroup.intersection

LGroup.**intersection** (*self*, *other*)

Returns (set) intersection of this label group and other.

In other words, this will return labels from this group which are also in other. Labels relative order will be kept intact, but only unique labels will be returned.

Parameters

other [Group or any sequence of labels] other labels

Returns

LSet

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a2')
>>> a['a0', 'a1'].intersection(a['a1', 'a2'])
a['a1'].set()
>>> a['a0', 'a1'].intersection('a1,a2')
a['a1'].set()
>>> a['a0', 'a1'].intersection(['a1', 'a2'])
a['a1'].set()
```

larray.LGroup.difference

LGroup.**difference** (*self*, *other*)

Returns (set) difference of this label group and other.

In other words, this will return labels from this group without those in other. Labels relative order will be kept intact, but only unique labels will be returned.

Parameters

other [Group or any sequence of labels] other labels

Returns

LSet

Examples

```
>>> from larray import Axis
>>> a = Axis('a=a0..a2')
>>> a['a0', 'a1'].difference(a['a1', 'a2'])
a['a0'].set()
>>> a['a0', 'a1'].difference('a1,a2')
a['a0'].set()
>>> a['a0', 'a1'].difference(['a1', 'a2'])
a['a0'].set()
```

larray.LGroup.containing

LGroup.**containing** (*self*, *substring*)

Returns a group with all the labels containing the specified substring.

Parameters

substring [str or Group] The substring to search for.

Returns

LGroup Group containing all the labels containing the substring.

Examples

```
>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
>>> group = people.startingwith('Bru')
>>> group
people['Bruce Wayne', 'Bruce Willis']
>>> group.containing('Will')
people['Bruce Willis']
```

larray.LGroup.startingwith

LGroup.**startingwith** (*self*, *prefix*)

Returns a group with the labels starting with the specified string.

Parameters

prefix [str or Group] The prefix to search for.

Returns

LGroup Group containing all the labels starting with the given string.

Examples

```
>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Arthur Dent', 'Harvey Dent'], 'people')
>>> group = people.endswith('Dent')
>>> group
people['Arthur Dent', 'Harvey Dent']
>>> group.startingwith('Art')
people['Arthur Dent']
```

larray.LGroup.endswith

`LGroup.endswith(self, suffix)`

Returns a group with the labels ending with the specified string.

Parameters

suffix [str or Group] The suffix to search for.

Returns

LGroup Group containing all the labels ending with the given string.

Examples

```
>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
>>> group = people.startingwith('Bru')
>>> group
people['Bruce Wayne', 'Bruce Willis']
>>> people.endswith('yne')
people['Bruce Wayne']
```

larray.LGroup.matching

`LGroup.matching(self, deprecated=None, pattern=None, regex=None)`

Returns a group with all the labels matching the specified pattern or regular expression.

Parameters

pattern [str or Group] Pattern to match.

- `?` matches any single character
- `*` matches any number of characters
- `[seq]` matches any character in seq
- `[!seq]` matches any character not in seq

To match any of the special characters above, wrap the character in brackets. For example, `[?]` matches the character `?`.

regex [str or Group] Regular expression pattern to match. Regular expressions are more powerful than what the simple patterns supported by the *pattern* argument but are also more complex to write. See [Regular Expression](#) for more details about how to build a regular expression pattern.

Returns

LGroup Group containing all the labels matching the pattern.

Examples

```
>>> from larray import Axis
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
```

Let us create a group with all names starting with B

```
>>> group = people.startingwith('B')
>>> group
people['Bruce Wayne', 'Bruce Willis']
```

Within that group, all labels containing any characters then W then any characters then s are given by

```
>>> group.matching(pattern='*W*s')
people['Bruce Willis']
```

Regular expressions are more powerful but usually harder to write and less readable. For example, here are the labels not containing the letter “i”.

```
>>> group.matching(regex='^[^i]*$')
people['Bruce Wayne']
```

larray.LGroup.to_hdf

`LGroup.to_hdf` (*self*, *filepath*, *key=None*, *axis_key=None*)

Writes group to a HDF file.

A HDF file can contain multiple groups. The ‘key’ parameter is a unique identifier for the group. The ‘axis_key’ parameter is the unique identifier for the associated axis. The associated axis will be saved if not already present in the HDF file.

Parameters

filepath [str] Path where the hdf file has to be written.

key [str or Group, optional] Key (path) of the group within the HDF file (see Notes below). If None, the name of the group is used. Defaults to None.

axis_key [str, optional] Key (path) of the associated axis in the HDF file (see Notes below). If None, the name of the axis associated with the group is used. Defaults to None.

Notes

Objects stored in a HDF file can be grouped together in *HDF groups*. If an object ‘my_obj’ is stored in a HDF group ‘my_group’, the key associated with this object is then ‘my_group/my_obj’. Be aware that a HDF group can have subgroups.

Examples

```
>>> from larray import Axis
>>> a = Axis("a=a0..a2")
>>> a.to_hdf('test.h5')
>>> a01 = a['a0,a1'] >> 'a01'
```

Save group

```
>>> # by default, the key is the name of the group
>>> # and axis_key the name of the associated axis
>>> a01.to_hdf('test.h5')
```

Save group with a specific key

```
>>> a01.to_hdf('test.h5', 'a_01')
```

Save group in a specific HDF group

```
>>> a.to_hdf('test.h5', 'groups/a01')
```

The associated axis is saved with the group if not already present in the HDF file

```
>>> b = Axis("b=b0..b2")
>>> b01 = b['b0,b1'] >> 'b01'
>>> # save both the group 'b01' and the associated axis 'b'
>>> b01.to_hdf('test.h5')
```

4.3.3 LSet

<code>LSet(key[, name, axis])</code>	Label set.
--------------------------------------	------------

larray.LSet

class `larray.LSet` (*key, name=None, axis=None*)

Label set.

Represents a set of (unique) labels of an axis.

Parameters

key [key] Anything usable for indexing. A key should be either sequence of labels, a slice with label bounds or a string.

name [str, optional] Name of the set.

axis [int, str, Axis, optional] Axis for set.

Examples

```
>>> from larray import Axis
>>> letters = Axis('letters=a..z')
>>> abc = letters[':c'].set() >> 'abc'
>>> abc
letters['a', 'b', 'c'].set() >> 'abc'
>>> abc & letters['b:d']
letters['b', 'c'].set()
```

__init__ (*self*, *key*, *name=None*, *axis=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self, key[, name, axis])</code>	Initialize self.
<code>by(self, length[, step, template])</code>	Split group into several groups of specified length.
<code>containing(self, substring)</code>	Returns a group with all the labels containing the specified substring.
<code>difference(self, other)</code>	
<code>endingwith(self, suffix)</code>	Returns a group with the labels ending with the specified string.
<code>equals(self, other)</code>	Checks if this group is equal to another group.
<code>eval(self)</code>	Translate key to labels, if it is not already, expanding slices in the process.
<code>intersection(self, other)</code>	
<code>matching(self[, deprecated, pattern, regex])</code>	Returns a group with all the labels matching the specified pattern or regular expression.
<code>named(self, name)</code>	Returns group with a different name.
<code>retarget_to(self, target_axis)</code>	Retarget group to another axis.
<code>set(self)</code>	Creates LSet from this group
<code>startingwith(self, prefix)</code>	Returns a group with the labels starting with the specified string.
<code>to_hdf(self, filepath[, key, axis_key])</code>	Writes group to a HDF file.
<code>to_label(self)</code>	Translate key to labels, if it is not already
<code>translate(self[, bound, stop])</code>	compute position(s) of group
<code>union(self, other)</code>	
<code>with_axis(self, axis)</code>	Returns group with a different axis.

Attributes

<code>axis</code>
<code>format_string</code>
<code>key</code>
<code>name</code>

4.3.4 AxisCollection

`AxisCollection([axes])`

`larray.AxisCollection`

class `larray.AxisCollection` (*axes=None*)

__init__ (*self, axes=None*)
 Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__(self[, axes])</code>	Initialize self.
<code>align(self, other[, join, axes])</code>	Align this axis collection with another.
<code>append(self, axis)</code>	Appends axis at the end of the collection.
<code>axis_id(self, axis)</code>	Returns the id of an axis.
<code>check_compatible(self, axes)</code>	Checks if axes passed as argument are compatible with those contained in the collection.
<code>combine_axes(self[, axes, sep, wildcard, ...])</code>	Combine several axes into one.
<code>copy(self)</code>	Returns a copy.
<code>extend(self, axes[, validate, replace_wildcards])</code>	Extends the collection by appending the axes from <i>axes</i> .
<code>get(self, key[, default, name])</code>	Returns axis corresponding to key.
<code>get_all(self, key)</code>	Returns all axes from key if present and length 1 wildcard axes otherwise.
<code>get_by_pos(self, key, i)</code>	Returns axis corresponding to a key, or to position <i>i</i> if the key has no name and key object not found.
<code>index(self, axis[, compatible])</code>	Returns the index of axis.
<code>insert(self, index, axis)</code>	Inserts axis before index.
<code>isaxis(self, value)</code>	Tests if input is an Axis object or the name of an axis contained in self.
<code>iter_labels(self[, axes, ascending])</code>	Returns a view of the axes labels.
<code>keys(self)</code>	Returns list of all axis names.
<code>pop(self[, axis])</code>	Removes and returns an axis.
<code>rename(self[, renames, to])</code>	Renames axes of the collection.
<code>replace(self[, axes_to_replace, new_axis, ...])</code>	Replace one, several or all axes of the collection.
<code>set_labels(self[, axis, labels, inplace])</code>	Replaces the labels of one or several axes.
<code>split_axes(self[, axes, sep, names, regex])</code>	Split axes and returns a new collection
<code>split_axis(*args, **kwargs)</code>	
<code>union(self, *args, **kwargs)</code>	
<code>without(self, axes)</code>	Returns a new collection without some axes.

Attributes

<i>display_names</i>	Returns the list of (display) names of the axes.
<i>ids</i>	Returns the list of ids of the axes.
<i>info</i>	Describes the collection (shape and labels for each axis).
<i>labels</i>	Returns the list of labels of the axes.
<i>names</i>	Returns the list of (raw) names of the axes.
<i>ndim</i>	
<i>shape</i>	Returns the shape of the collection.
<i>size</i>	Returns the size of the collection, i.e.

<i>AxisCollection.names</i>	Returns the list of (raw) names of the axes.
<i>AxisCollection.display_names</i>	Returns the list of (display) names of the axes.
<i>AxisCollection.labels</i>	Returns the list of labels of the axes.
<i>AxisCollection.shape</i>	Returns the shape of the collection.
<i>AxisCollection.size</i>	Returns the size of the collection, i.e.
<i>AxisCollection.info</i>	Describes the collection (shape and labels for each axis).
<i>AxisCollection.copy(self)</i>	Returns a copy.

`larray.AxisCollection.names`

property `AxisCollection.names`

Returns the list of (raw) names of the axes.

Returns

list List of names of the axes.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> AxisCollection([age, sex, time]).names
['age', 'sex', 'time']
```

`larray.AxisCollection.display_names`

property `AxisCollection.display_names`

Returns the list of (display) names of the axes.

Returns

list List of names of the axes. Wildcard axes are displayed with an attached *. Anonymous axes (name = None) are replaced by their position wrapped in braces.

Examples

```
>>> a = Axis(['a1', 'a2'], 'a')
>>> b = Axis(2, 'b')
>>> c = Axis(['c1', 'c2'])
>>> d = Axis(3)
>>> AxisCollection([a, b, c, d]).display_names
['a', 'b*', '{2}', '{3}*']
```

larray.AxisCollection.labels

property AxisCollection.labels

Returns the list of labels of the axes.

Returns

list List of labels of the axes.

Examples

```
>>> age = Axis(range(10), 'age')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> AxisCollection([age, time]).labels
[array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 array([2007, 2008, 2009, 2010])]
```

larray.AxisCollection.shape

property AxisCollection.shape

Returns the shape of the collection.

Returns

tuple Tuple of lengths of axes.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> AxisCollection([age, sex, time]).shape
(20, 2, 4)
```

larray.AxisCollection.size

property `AxisCollection.size`

Returns the size of the collection, i.e. the number of elements of the array.

Returns

int Number of elements of the array.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> AxisCollection([age, sex, time]).size
160
```

larray.AxisCollection.info

property `AxisCollection.info`

Describes the collection (shape and labels for each axis).

Returns

str Description of the AxisCollection (shape and labels for each axis).

Examples

```
>>> age = Axis(20, 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> AxisCollection([age, sex, time]).info
20 x 2 x 4
age* [20]: 0 1 2 ... 17 18 19
sex [2]: 'M' 'F'
time [4]: 2007 2008 2009 2010
```

larray.AxisCollection.copy

`AxisCollection.copy(self)`

Returns a copy.

Searching

<code>AxisCollection.keys(self)</code>	Returns list of all axis names.
<code>AxisCollection.index(self, axis[, compatible])</code>	Returns the index of axis.
<code>AxisCollection.axis_id(self, axis)</code>	Returns the id of an axis.
<code>AxisCollection.ids</code>	Returns the list of ids of the axes.
<code>AxisCollection.iter_labels(self, axes, ...)</code>	Returns a view of the axes labels.

larray.AxisCollection.keys

`AxisCollection.keys` (*self*)
Returns list of all axis names.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> AxisCollection([age, sex, time]).keys()
['age', 'sex', 'time']
```

larray.AxisCollection.index

`AxisCollection.index` (*self, axis, compatible=False*)
Returns the index of axis.

axis can be a name or an Axis object (or an index). If the Axis object itself exists in the list, `index()` will return it. Otherwise, it will return the index of the local axis with the same name than the key (whether it is compatible or not).

Parameters

axis [Axis or int or str] Can be the axis itself or its position (returned if represents a valid index) or its name.

compatible [bool, optional] If axis is an Axis, whether to find an exact match (using `Axis.equals`) or any compatible axis (using `Axis.iscompatible`)

Returns

int Index of the axis.

Raises

ValueError Raised if the axis is not present.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> col = AxisCollection([age, sex, time])
>>> col.index(time)
2
>>> col.index('sex')
1
```

larray.AxisCollection.axis_id

AxisCollection.**axis_id**(*self*, *axis*)

Returns the id of an axis.

Returns

str or int Id of axis, which is its name if defined and its position otherwise.

Examples

```
>>> a = Axis(2, 'a')
>>> b = Axis(2)
>>> c = Axis(2, 'c')
>>> col = AxisCollection([a, b, c])
>>> col.axis_id(a)
'a'
>>> col.axis_id(b)
1
>>> col.axis_id(c)
'c'
```

larray.AxisCollection.ids

property AxisCollection.**ids**

Returns the list of ids of the axes.

Returns

list List of ids of the axes.

See also:

[*axis_id*](#)

Examples

```
>>> a = Axis(2, 'a')
>>> b = Axis(2)
>>> c = Axis(2, 'c')
>>> AxisCollection([a, b, c]).ids
['a', 1, 'c']
```

larray.AxisCollection.iter_labels

AxisCollection.**iter_labels**(*self*, *axes=None*, *ascending=True*)

Returns a view of the axes labels.

Parameters

axes [int, str or Axis or tuple of them, optional] Axis or axes along which to iterate and in which order. Defaults to None (all axes in the order they are in the collection).

ascending [bool, optional] Whether or not to iterate the axes in ascending order (from start to end). Defaults to True.

Returns

Sequence An object you can iterate (loop) on and index by position.

Examples

```
>>> from larray import ndtest
>>> axes = ndtest((2, 2)).axes
>>> axes
AxisCollection([
    Axis(['a0', 'a1'], 'a'),
    Axis(['b0', 'b1'], 'b')
])
>>> axes.iter_labels()[0]
(a.i[0], b.i[0])
>>> for index in axes.iter_labels():
...     print(index)
(a.i[0], b.i[0])
(a.i[0], b.i[1])
(a.i[1], b.i[0])
(a.i[1], b.i[1])
>>> axes.iter_labels(ascending=False)[0]
(a.i[1], b.i[1])
>>> for index in axes.iter_labels(ascending=False):
...     print(index)
(a.i[1], b.i[1])
(a.i[1], b.i[0])
(a.i[0], b.i[1])
(a.i[0], b.i[0])
>>> axes.iter_labels(('b', 'a'))[0]
(b.i[0], a.i[0])
>>> for index in axes.iter_labels(('b', 'a')):
...     print(index)
(b.i[0], a.i[0])
(b.i[0], a.i[1])
(b.i[1], a.i[0])
(b.i[1], a.i[1])
>>> axes.iter_labels('b')[0]
(b.i[0],)
>>> for index in axes.iter_labels('b'):
...     print(index)
(b.i[0],)
(b.i[1],)
```

Modifying/Selecting

<code>AxisCollection.get(self, key[, default, name])</code>	Returns axis corresponding to key.
<code>AxisCollection.get_by_pos(self, key, i)</code>	Returns axis corresponding to a key, or to position <i>i</i> if the key has no name and key object not found.
<code>AxisCollection.get_all(self, key)</code>	Returns all axes from key if present and length 1 wild-card axes otherwise.
<code>AxisCollection.pop(self[, axis])</code>	Removes and returns an axis.
<code>AxisCollection.append(self, axis)</code>	Appends axis at the end of the collection.
<code>AxisCollection.extend(self, axes[, ...])</code>	Extends the collection by appending the axes from <i>axes</i> .
<code>AxisCollection.insert(self, index, axis)</code>	Inserts axis before index.
<code>AxisCollection.rename(self[, renames, to])</code>	Renames axes of the collection.
<code>AxisCollection.replace(self[, ...])</code>	Replace one, several or all axes of the collection.
<code>AxisCollection.set_labels(self[, axis, ...])</code>	Replaces the labels of one or several axes.
<code>AxisCollection.without(self, axes)</code>	Returns a new collection without some axes.
<code>AxisCollection.combine_axes(self[, axes, ...])</code>	Combine several axes into one.
<code>AxisCollection.split_axes(self[, axes, sep, ...])</code>	Split axes and returns a new collection
<code>AxisCollection.align(self, other[, join, axes])</code>	Align this axis collection with another.

larray.AxisCollection.get

`AxisCollection.get(self, key, default=None, name=None)`

Returns axis corresponding to key. If not found, the argument *name* is used to create a new Axis. If *name* is None, the *default* axis is then returned.

Parameters

key [key] Key corresponding to an axis of the current AxisCollection.

default [axis, optional] Default axis to return if key doesn't correspond to any axis of the collection and argument *name* is None.

name [str, optional] If key doesn't correspond to any axis of the collection, a new Axis with this name is created and returned.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> col = AxisCollection([age, time])
>>> col.get('time')
Axis([2007, 2008, 2009, 2010], 'time')
>>> col.get('sex', sex)
Axis(['M', 'F'], 'sex')
>>> col.get('nb_children', None, 'nb_children')
Axis(1, 'nb_children')
```

`larray.AxisCollection.get_by_pos`

`AxisCollection.get_by_pos` (*self*, *key*, *i*)

Returns axis corresponding to a key, or to position *i* if the key has no name and key object not found.

Parameters

key [key] Key corresponding to an axis.

i [int] Position of the axis (used only if search by key failed).

Returns

Axis Axis corresponding to the key or the position *i*.

Examples

```

>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> col = AxisCollection([age, sex, time])
>>> col.get_by_pos('sex', 1)
Axis(['M', 'F'], 'sex')

```

`larray.AxisCollection.get_all`

`AxisCollection.get_all` (*self*, *key*)

Returns all axes from key if present and length 1 wildcard axes otherwise.

Parameters

key [AxisCollection]

Returns

AxisCollection

Raises

AssertionError Raised if the input key is not an AxisCollection object.

Examples

```

>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> city = Axis(['London', 'Paris', 'Rome'], 'city')
>>> col = AxisCollection([age, sex, time])
>>> col2 = AxisCollection([age, city, time])
>>> col.get_all(col2)
AxisCollection([
    Axis([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    ↪ 'age'),
    Axis(1, 'city'),
    Axis([2007, 2008, 2009, 2010], 'time')
])

```

larray.AxisCollection.pop

AxisCollection.**pop**(*self*, *axis=-1*)

Removes and returns an axis.

Parameters

axis [key, optional] Axis to remove and return. Default value is -1 (last axis).

Returns

Axis If no argument is provided, the last axis is removed and returned.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> col = AxisCollection([age, sex, time])
>>> col.pop('age')
Axis([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19], 'age
→')
>>> col
AxisCollection([
  Axis(['M', 'F'], 'sex'),
  Axis([2007, 2008, 2009, 2010], 'time')
])
>>> col.pop()
Axis([2007, 2008, 2009, 2010], 'time')
```

larray.AxisCollection.append

AxisCollection.**append**(*self*, *axis*)

Appends axis at the end of the collection.

Parameters

axis [Axis] Axis to append.

Examples

```
>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> col = AxisCollection([age, sex])
>>> col.append(time)
>>> col
AxisCollection([
  Axis([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
→'age'),
  Axis(['M', 'F'], 'sex'),
  Axis([2007, 2008, 2009, 2010], 'time')
])
```


larray.AxisCollection.extend

`AxisCollection.extend(self, axes, validate=True, replace_wildcards=False)`

Extends the collection by appending the axes from *axes*.

Parameters

axes [sequence of Axis (list, tuple, AxisCollection)]

validate [bool, optional]

replace_wildcards [bool, optional]

Raises

TypeError Raised if *axes* is not a sequence of Axis (list, tuple or AxisCollection)

Examples

```

>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> col = AxisCollection(age)
>>> col.extend([sex, time])
>>> col
AxisCollection([
    Axis([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    ↪ 'age'),
    Axis(['M', 'F'], 'sex'),
    Axis([2007, 2008, 2009, 2010], 'time')
])

```

larray.AxisCollection.insert

`AxisCollection.insert(self, index, axis)`

Inserts axis before index.

Parameters

index [int] position of the inserted axis.

axis [Axis] axis to insert.

Examples

```

>>> age = Axis(range(20), 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009, 2010], 'time')
>>> col = AxisCollection([age, time])
>>> col.insert(1, sex)
>>> col
AxisCollection([
    Axis([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    ↪ 'age'),
    Axis(['M', 'F'], 'sex'),

```

(continues on next page)

(continued from previous page)

```
Axis([2007, 2008, 2009, 2010], 'time')
])
```

larray.AxisCollection.rename

`AxisCollection.rename` (*self*, *renames=None*, *to=None*, ***kwargs*)

Renames axes of the collection.

Parameters

renames [axis ref or dict {axis ref: str} or list of tuple (axis ref, str), optional] Renames to apply. If a single axis reference is given, the *to* argument must be used.

to [str or Axis, optional] New name if *renames* contains a single axis reference.

****kwargs** [str or Axis] New name for each axis given as a keyword argument.

Returns

AxisCollection collection with axes renamed.

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> axes = AxisCollection([nat, sex])
>>> axes
AxisCollection([
  Axis(['BE', 'FO'], 'nat'),
  Axis(['M', 'F'], 'sex')
])
>>> axes.rename(nat, 'nat2')
AxisCollection([
  Axis(['BE', 'FO'], 'nat2'),
  Axis(['M', 'F'], 'sex')
])
>>> axes.rename(nat='nat2', sex='sex2')
AxisCollection([
  Axis(['BE', 'FO'], 'nat2'),
  Axis(['M', 'F'], 'sex2')
])
>>> axes.rename([('nat', 'nat2'), ('sex', 'sex2')])
AxisCollection([
  Axis(['BE', 'FO'], 'nat2'),
  Axis(['M', 'F'], 'sex2')
])
>>> axes.rename({'nat': 'nat2', 'sex': 'sex2'})
AxisCollection([
  Axis(['BE', 'FO'], 'nat2'),
  Axis(['M', 'F'], 'sex2')
])
```

larray.AxisCollection.replace

`AxisCollection.replace` (*self*, *axes_to_replace*=None, *new_axis*=None, *inplace*=False, ***kwargs*)

Replace one, several or all axes of the collection.

Parameters

axes_to_replace [axis ref or dict {axis ref: axis} or list of tuple (axis ref, axis)]

or list of Axis or AxisCollection, optional

Axes to replace. If a single axis reference is given, the *new_axis* argument must be provided. If a list of Axis or an AxisCollection is given, all axes will be replaced by the new ones. In that case, the number of new axes must match the number of the old ones. Defaults to None.

new_axis [axis ref, optional] New axis if *axes_to_replace* contains a single axis reference. Defaults to None.

inplace [bool, optional] Whether or not to modify the original object or return a new AxisCollection and leave the original intact. Defaults to False.

****kwargs** [Axis] New axis for each axis to replace given as a keyword argument.

Returns

AxisCollection AxisCollection with axes replaced.

Examples

```
>>> from larray import ndtest
>>> axes = ndtest((2, 3)).axes
>>> axes
AxisCollection([
  Axis(['a0', 'a1'], 'a'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
>>> row = Axis(['r0', 'r1'], 'row')
>>> column = Axis(['c0', 'c1', 'c2'], 'column')
```

Replace one axis (second argument *new_axis* must be provided)

```
>>> axes.replace(X.a, row)
>>> # or
>>> axes.replace(X.a, "row=r0,r1")
AxisCollection([
  Axis(['r0', 'r1'], 'row'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
```

Replace several axes (keywords, list of tuple or dictionary)

```
>>> axes.replace(a=row, b=column)
>>> # or
>>> axes.replace(a="row=r0,r1", b="column=c0,c1,c2")
>>> # or
>>> axes.replace([(X.a, row), (X.b, column)])
>>> # or
>>> axes.replace({X.a: row, X.b: column})
AxisCollection([
```

(continues on next page)

(continued from previous page)

```

    Axis(['r0', 'r1'], 'row'),
    Axis(['c0', 'c1', 'c2'], 'column')
])

```

Replace all axes (list of axes or AxisCollection)

```

>>> axes.replace([row, column])
AxisCollection([
    Axis(['r0', 'r1'], 'row'),
    Axis(['c0', 'c1', 'c2'], 'column')
])
>>> arr = ndtest([row, column])
>>> axes.replace(arr.axes)
AxisCollection([
    Axis(['r0', 'r1'], 'row'),
    Axis(['c0', 'c1', 'c2'], 'column')
])

```

larray.AxisCollection.set_labels

AxisCollection.**set_labels** (*self*, *axis=None*, *labels=None*, *inplace=False*, ***kwargs*)

Replaces the labels of one or several axes.

Parameters

axis [string or Axis or dict] Axis for which we want to replace labels, or mapping {axis: changes} where changes can either be the complete list of labels, a mapping {old_label: new_label} or a function to transform labels. If there is no ambiguity (two or more axes have the same labels), *axis* can be a direct mapping {old_label: new_label}.

labels [int, str, iterable or mapping or function, optional] Integer or list of values usable as the collection of labels for an Axis. If this is mapping, it must be {old_label: new_label}. If it is a function, it must be a function accepting a single argument (a label) and returning a single value. This argument must not be used if *axis* is a mapping.

inplace [bool, optional] Whether or not to modify the original object or return a new AxisCollection and leave the original intact. Defaults to False.

****kwargs** : *axis*='labels for each axis you want to set labels.

Returns

AxisCollection AxisCollection with modified labels.

Examples

```

>>> from larray import ndtest
>>> axes = AxisCollection('nat=BE,FO;sex=M,F')
>>> axes
AxisCollection([
    Axis(['BE', 'FO'], 'nat'),
    Axis(['M', 'F'], 'sex')
])
>>> axes.set_labels('sex', ['Men', 'Women'])
AxisCollection([

```

(continues on next page)

(continued from previous page)

```

    Axis(['BE', 'FO'], 'nat'),
    Axis(['Men', 'Women'], 'sex')
])

```

when passing a single string as labels, it will be interpreted to create the list of labels, so that one can use the same syntax than during axis creation.

```

>>> axes.set_labels('sex', 'Men,Women')
AxisCollection([
    Axis(['BE', 'FO'], 'nat'),
    Axis(['Men', 'Women'], 'sex')
])

```

to replace only some labels, one must give a mapping giving the new label for each label to replace

```

>>> axes.set_labels('sex', {'M': 'Men'})
AxisCollection([
    Axis(['BE', 'FO'], 'nat'),
    Axis(['Men', 'F'], 'sex')
])

```

to transform labels by a function, use any function accepting and returning a single argument:

```

>>> axes.set_labels('nat', str.lower)
AxisCollection([
    Axis(['be', 'fo'], 'nat'),
    Axis(['M', 'F'], 'sex')
])

```

to replace labels for several axes at the same time, one should give a mapping giving the new labels for each changed axis

```

>>> axes.set_labels({'sex': 'Men,Women', 'nat': 'Belgian,Foreigner'})
AxisCollection([
    Axis(['Belgian', 'Foreigner'], 'nat'),
    Axis(['Men', 'Women'], 'sex')
])

```

or use keyword arguments

```

>>> axes.set_labels(sex='Men,Women', nat='Belgian,Foreigner')
AxisCollection([
    Axis(['Belgian', 'Foreigner'], 'nat'),
    Axis(['Men', 'Women'], 'sex')
])

```

one can also replace some labels in several axes by giving a mapping of mappings

```

>>> axes.set_labels({'sex': {'M': 'Men'}, 'nat': {'BE': 'Belgian'}})
AxisCollection([
    Axis(['Belgian', 'FO'], 'nat'),
    Axis(['Men', 'F'], 'sex')
])

```

when there is no ambiguity (two or more axes have the same labels), it is possible to give a mapping between old and new labels

```
>>> axes.set_labels({'M': 'Men', 'BE': 'Belgian'})
AxisCollection([
  Axis(['Belgian', 'FO'], 'nat'),
  Axis(['Men', 'F'], 'sex')
])
```

larray.AxisCollection.without

AxisCollection.**without** (*self*, *axes*)

Returns a new collection without some axes.

You can use a comma separated list of names.

Parameters

axes [int, str, Axis or sequence of those] Axes to not include in the returned AxisCollection. In case of string, axes are separated by a comma and no whitespace is accepted.

Returns

AxisCollection New collection without some axes.

Notes

Set operation so axes can contain axes not present in self

Examples

```
>>> age = Axis('age=0..5')
>>> sex = Axis('sex=M,F')
>>> time = Axis('time=2015..2017')
>>> col = AxisCollection([age, sex, time])
>>> col.without([age, sex])
AxisCollection([
  Axis([2015, 2016, 2017], 'time')
])
>>> col.without(0)
AxisCollection([
  Axis(['M', 'F'], 'sex'),
  Axis([2015, 2016, 2017], 'time')
])
>>> col.without('sex,time')
AxisCollection([
  Axis([0, 1, 2, 3, 4, 5], 'age')
])
```

larray.AxisCollection.combine_axes

`AxisCollection.combine_axes` (*self*, *axes=None*, *sep='_'*, *wildcard=False*, *front_if_spread=False*)
 Combine several axes into one.

Parameters

axes [tuple, list, AxisCollection of axes or list of combination of those or dict, optional] axes to combine. Tuple, list or AxisCollection will combine several axes into one. To chain several axes combinations, pass a list of tuple/list/AxisCollection of axes. To set the name(s) of resulting axis(es), use a {(axes, to, combine): 'new_axis_name'} dictionary. Defaults to all axes.

sep [str, optional] delimiter to use for combining. Defaults to '_'.

wildcard [bool, optional] whether or not to produce a wildcard axis even if the axes to combine are not. This is much faster, but loose axes labels.

front_if_spread [bool, optional] whether or not to move the combined axis at the front (it will be the first axis) if the combined axes are not next to each other.

Returns

AxisCollection New AxisCollection with combined axes.

Examples

```
>>> axes = AxisCollection('a=a0,a1;b=b0..b2')
>>> axes
AxisCollection([
  Axis(['a0', 'a1'], 'a'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
>>> axes.combine_axes()
AxisCollection([
  Axis(['a0_b0', 'a0_b1', 'a0_b2', 'a1_b0', 'a1_b1', 'a1_b2'], 'a_b')
])
>>> axes.combine_axes(sep='/')
AxisCollection([
  Axis(['a0/b0', 'a0/b1', 'a0/b2', 'a1/b0', 'a1/b1', 'a1/b2'], 'a/b')
])
>>> axes += AxisCollection('c=c0..c2;d=d0,d1')
>>> axes.combine_axes({'a', 'c'})
AxisCollection([
  Axis(['a0_c0', 'a0_c1', 'a0_c2', 'a1_c0', 'a1_c1', 'a1_c2'], 'a_c'),
  Axis(['b0', 'b1', 'b2'], 'b'),
  Axis(['d0', 'd1'], 'd')
])
>>> axes.combine_axes({'a', 'c': 'ac'})
AxisCollection([
  Axis(['a0_c0', 'a0_c1', 'a0_c2', 'a1_c0', 'a1_c1', 'a1_c2'], 'ac'),
  Axis(['b0', 'b1', 'b2'], 'b'),
  Axis(['d0', 'd1'], 'd')
])
```

make several combinations at once

```
>>> axes.combine_axes([('a', 'c'), ('b', 'd')])
AxisCollection([
  Axis(['a0_c0', 'a0_c1', 'a0_c2', 'a1_c0', 'a1_c1', 'a1_c2'], 'a_c'),
  Axis(['b0_d0', 'b0_d1', 'b1_d0', 'b1_d1', 'b2_d0', 'b2_d1'], 'b_d')
])
>>> axes.combine_axes({'a', 'c': 'ac', ('b', 'd'): 'bd'})
AxisCollection([
  Axis(['a0_c0', 'a0_c1', 'a0_c2', 'a1_c0', 'a1_c1', 'a1_c2'], 'ac'),
  Axis(['b0_d0', 'b0_d1', 'b1_d0', 'b1_d1', 'b2_d0', 'b2_d1'], 'bd')
])
```

`larray.AxisCollection.split_axes`

`AxisCollection.split_axes` (*self*, *axes=None*, *sep='_'*, *names=None*, *regex=None*)

Split axes and returns a new collection

The split axes are inserted where the combined axis was.

Parameters

axes [int, str, Axis or any combination of those, optional] axes to split. All labels *must* contain the given delimiter string. To split several axes at once, pass a list or tuple of axes to split. To set the names of resulting axes, use a {'axis_to_split': (new, axes)} dictionary. Defaults to all axes whose name contains the *sep* delimiter.

sep [str, optional] delimiter to use for splitting. Defaults to '_'. When *regex* is provided, the delimiter is only used on *names* if given as one string or on axis name if *names* is None.

names [str or list of str, optional] names of resulting axes. Defaults to None.

regex [str, optional] use the *regex* regular expression to split labels instead of the *sep* delimiter. Defaults to None.

Returns

AxisCollection

See also:

[`Axis.split`](#)

[`Array.split_axes`](#)

Examples

```
>>> col = AxisCollection('a=a0,a1;b=b0..b2')
>>> col
AxisCollection([
  Axis(['a0', 'a1'], 'a'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
>>> combined = col.combine_axes()
>>> combined
AxisCollection([
  Axis(['a0_b0', 'a0_b1', 'a0_b2', 'a1_b0', 'a1_b1', 'a1_b2'], 'a_b')
])
>>> combined.split_axes()
```

(continues on next page)

(continued from previous page)

```
AxisCollection([
    Axis(['a0', 'a1'], 'a'),
    Axis(['b0', 'b1', 'b2'], 'b')
])
```

Split labels using a regular expression

```
>>> combined = AxisCollection('a_b = a0b0..alb2')
>>> combined
AxisCollection([
    Axis(['a0b0', 'a0b1', 'a0b2', 'alb0', 'alb1', 'alb2'], 'a_b')
])
>>> combined.split_axes('a_b', regex=r'(\w{2})(\w{2})')
AxisCollection([
    Axis(['a0', 'a1'], 'a'),
    Axis(['b0', 'b1', 'b2'], 'b')
])
```

Split several axes at once

```
>>> combined = AxisCollection('a_b = a0_b0..a1_b1; c_d = c0_d0..c1_d1')
>>> combined
AxisCollection([
    Axis(['a0_b0', 'a0_b1', 'a1_b0', 'a1_b1'], 'a_b'),
    Axis(['c0_d0', 'c0_d1', 'c1_d0', 'c1_d1'], 'c_d')
])
>>> # equivalent to combined.split_axes() which split all axes
>>> # containing the delimiter defined by the argument `sep`
>>> combined.split_axes(['a_b', 'c_d'])
AxisCollection([
    Axis(['a0', 'a1'], 'a'),
    Axis(['b0', 'b1'], 'b'),
    Axis(['c0', 'c1'], 'c'),
    Axis(['d0', 'd1'], 'd')
])
>>> combined.split_axes({'a_b': ('A', 'B'), 'c_d': ('C', 'D')})
AxisCollection([
    Axis(['a0', 'a1'], 'A'),
    Axis(['b0', 'b1'], 'B'),
    Axis(['c0', 'c1'], 'C'),
    Axis(['d0', 'd1'], 'D')
])
```

larray.AxisCollection.align

`AxisCollection.align` (*self*, *other*, *join*='outer', *axes*=None)

Align this axis collection with another.

This ensures all common axes are compatible.

Parameters

other [AxisCollection]

join [{‘outer’, ‘inner’, ‘left’, ‘right’, ‘exact’}, optional] Defaults to ‘outer’.

axes [AxisReference or sequence of them, optional] Axes to align. Need to be valid in both arrays. Defaults to None (all common axes). This must be specified when mixing anonymous and non-anonymous axes.

Returns

(left, right) [(AxisCollection, AxisCollection)] Aligned collections

See also:

Array.align

Examples

```
>>> col1 = AxisCollection("a=a0..a1;b=b0..b2")
>>> col1
AxisCollection([
  Axis(['a0', 'a1'], 'a'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
>>> col2 = AxisCollection("a=a0..a2;c=c0..c0;b=b0..b1")
>>> col2
AxisCollection([
  Axis(['a0', 'a1', 'a2'], 'a'),
  Axis(['c0'], 'c'),
  Axis(['b0', 'b1'], 'b')
])
>>> aligned1, aligned2 = col1.align(col2)
>>> aligned1
AxisCollection([
  Axis(['a0', 'a1', 'a2'], 'a'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
>>> aligned2
AxisCollection([
  Axis(['a0', 'a1', 'a2'], 'a'),
  Axis(['c0'], 'c'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
```

Using anonymous axes

```
>>> col1 = AxisCollection("a0..a1;b0..b2")
>>> col1
AxisCollection([
  Axis(['a0', 'a1'], None),
  Axis(['b0', 'b1', 'b2'], None)
])
>>> col2 = AxisCollection("a0..a2;b0..b1;c0..c0")
>>> col2
AxisCollection([
  Axis(['a0', 'a1', 'a2'], None),
  Axis(['b0', 'b1'], None),
  Axis(['c0'], None)
])
>>> aligned1, aligned2 = col1.align(col2)
>>> aligned1
```

(continues on next page)

(continued from previous page)

```

AxisCollection([
    Axis(['a0', 'a1', 'a2'], None),
    Axis(['b0', 'b1', 'b2'], None)
])
>>> aligned2
AxisCollection([
    Axis(['a0', 'a1', 'a2'], None),
    Axis(['b0', 'b1', 'b2'], None),
    Axis(['c0'], None)
])

```

Testing

<code>AxisCollection.isaxis(self, value)</code>	Tests if input is an Axis object or the name of an axis contained in self.
<code>AxisCollection.check_compatible(self, axes)</code>	Checks if axes passed as argument are compatible with those contained in the collection.

larray.AxisCollection.isaxis

`AxisCollection.isaxis(self, value)`

Tests if input is an Axis object or the name of an axis contained in self.

Parameters

value [Axis or str] Input axis or string

Returns

bool True if input is an Axis object or the name of an axis contained in the current `AxisCollection` instance, False otherwise.

Examples

```

>>> a = Axis('a=a0,a1')
>>> b = Axis('b=b0,b1')
>>> col = AxisCollection([a, b])
>>> col.isaxis(a)
True
>>> col.isaxis('b')
True
>>> col.isaxis('c')
False

```

larray.AxisCollection.check_compatible

`AxisCollection.check_compatible` (*self*, *axes*)

Checks if axes passed as argument are compatible with those contained in the collection. Raises `ValueError` if not.

See also:

[Axis.iscompatible](#)

4.3.5 Array

- *Overview*
- *Array Creation Functions*
- *Copying*
- *Inspecting*
- *Modifying/Selecting*
- *Changing Axes or Labels*
- *Aggregation Functions*
- *Sorting*
- *Reshaping/Extending/Reordering*
- *Testing/Searching*
- *Iterating*
- *Operators*
- *Miscellaneous*
- *Converting to Pandas objects*
- *Plotting*

Overview

<code>Array</code> (<i>data</i> [, <i>axes</i> , <i>title</i> , <i>meta</i> , <i>dtype</i>])	An Array object represents a multidimensional, homogeneous array of fixed-size items with labeled axes.
--	---

larray.Array

class `larray.Array` (*data*, *axes=None*, *title=None*, *meta=None*, *dtype=None*)

An Array object represents a multidimensional, homogeneous array of fixed-size items with labeled axes.

The function `asarray()` can be used to convert a NumPy array or Pandas DataFrame into an Array.

Parameters

data [scalar, tuple, list or NumPy ndarray] Input data.

axes [collection (tuple, list or AxisCollection) of axes (int, str or Axis), optional] Axes.

title [str, optional] Deprecated. See ‘meta’ below.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

dtype [type, optional] Datatype for the array. Defaults to None (inferred from the data).

Warning: Metadata is not kept when actions or methods are applied on an array except for operations modifying the object in-place, such as: `pop[age < 10] = 0`. Do not add metadata to an array if you know you will apply actions or methods on it before dumping it.

See also:

sequence Create an Array by sequentially applying modifications to the array along axis.

ndtest Create a test Array with increasing elements.

zeros Create an Array, each element of which is zero.

ones Create an Array, each element of which is 1.

full Create an Array filled with a given value.

empty Create an Array, but leave its allocated memory unchanged (i.e., it contains “garbage”).

Examples

```
>>> age = Axis([10, 11, 12], 'age')
>>> sex = Axis('sex=M,F')
>>> time = Axis([2007, 2008, 2009], 'time')
>>> axes = [age, sex, time]
>>> data = np.zeros((len(axes), len(sex), len(time)))
```

```
>>> Array(data, axes)
age  sex\time  2007  2008  2009
10      M    0.0   0.0   0.0
10      F    0.0   0.0   0.0
11      M    0.0   0.0   0.0
11      F    0.0   0.0   0.0
12      M    0.0   0.0   0.0
12      F    0.0   0.0   0.0
>>> # with metadata (Python <= 3.5)
>>> arr = Array(data, axes, meta=[('title', 'my title'), ('author', 'John Smith
↳')])
>>> # with metadata (Python 3.6+)
>>> arr = Array(data, axes, meta=Metadata(title='my title', author='John Smith'))
```

Array creation functions

```
>>> full(axes, 10.0)
age  sex\time  2007  2008  2009
10      M   10.0  10.0  10.0
10      F   10.0  10.0  10.0
11      M   10.0  10.0  10.0
11      F   10.0  10.0  10.0
```

(continues on next page)

(continued from previous page)

```

12      M  10.0  10.0  10.0
12      F  10.0  10.0  10.0
>>> arr = empty(axes)
>>> arr['F'] = 1.0
>>> arr['M'] = -1.0
>>> arr
age  sex\time  2007  2008  2009
10      M  -1.0  -1.0  -1.0
10      F   1.0   1.0   1.0
11      M  -1.0  -1.0  -1.0
11      F   1.0   1.0   1.0
12      M  -1.0  -1.0  -1.0
12      F   1.0   1.0   1.0
>>> bysex = sequence(sex, initial=-1, inc=2)
>>> bysex
sex  M  F
    -1  1
>>> sequence(age, initial=10, inc=bysex)
sex\age  10  11  12
      M  10   9   8
      F  10  11  12

```

Attributes

data [NumPy ndarray] Data.

axes [AxisCollection] Axes.

meta [Metadata] Returns metadata of the array.

__init__ (*self*, *data*, *axes=None*, *title=None*, *meta=None*, *dtype=None*)
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self, data[, axes, title, meta, dtype])</code>	Initialize self.
<code>align(self, other[, join, fill_value, axes])</code>	Align two arrays on their axes with the specified join method.
<code>all(*axes_and_groups[, out, skipna, keepaxes])</code>	Test whether all selected elements evaluate to True.
<code>all_by(*axes_and_groups[, out, skipna, keep-axes])</code>	Test whether all selected elements evaluate to True.
<code>any(*axes_and_groups[, out, skipna, keepaxes])</code>	Test whether any selected elements evaluate to True.
<code>any_by(*axes_and_groups[, out, skipna, keep-axes])</code>	Test whether any selected elements evaluate to True.
<code>append(self, axis, value[, label])</code>	Adds an array to self along an axis.
<code>apply(self, transform, *args, **kwargs)</code>	Apply a transformation function to array elements.
<code>apply_map(self, mapping[, dtype])</code>	Apply a transformation mapping to array elements.
<code>argmax(*args, **kwargs)</code>	
<code>argmin(*args, **kwargs)</code>	
<code>argsort(*args, **kwargs)</code>	
<code>as_table(self[, maxlines, edgeitems, light, ...])</code>	Deprecated.
<code>astype(dtype[, order, casting, subok, copy])</code>	Copy of the array, cast to a specified type.

Continued on next page

Table 28 – continued from previous page

<i>broadcast_with</i> (self, target)	Returns an array that is (NumPy) broadcastable with target.
<i>clip</i> (self[, minval, maxval, out])	Clip (limit) the values in an array.
<i>combine_axes</i> (self[, axes, sep, wildcard])	Combine several axes into one.
<i>compact</i> (self)	Detects and removes “useless” axes (ie axes for which values are constant over the whole axis)
<i>copy</i> (self)	Returns a copy of the array.
<i>cumprod</i> (self[, axis])	Returns the cumulative product of array elements.
<i>cumsum</i> (self[, axis])	Returns the cumulative sum of array elements along an axis.
<i>describe</i> (self, *args, **kwargs)	Descriptive summary statistics, excluding NaN values.
<i>describe_by</i> (self, *args, **kwargs)	Descriptive summary statistics, excluding NaN values, along axes or for groups.
<i>diff</i> (self[, axis, d, n, label])	Calculates the n-th order discrete difference along a given axis.
<i>divnot0</i> (self, other)	Divides array by other, but returns 0.0 where other is 0.
<i>drop</i> (self[, labels])	Return array without some labels or indices along an axis.
<i>drop_labels</i> (*args, **kwargs)	
<i>dump</i> (self[, header, wide, value_name, ...])	Dump array as a 2D nested list.
<i>eq</i> (self, other[, rtol, atol, nans_equal])	Compares self with another array element-wise and returns an array of booleans.
<i>equals</i> (self, other[, rtol, atol, ...])	Compares self with another array and returns True if they have the same axes and elements, False otherwise.
<i>expand</i> (self[, target_axes, out, readonly])	Expands array to target_axes.
<i>extend</i> (self, axis, other)	Adds an array to self along an axis.
<i>filter</i> (self[, collapse])	Filters the array along the axes given as keyword arguments.
<i>growth_rate</i> (self[, axis, d, label])	Calculates the growth along a given axis.
<i>ignore_labels</i> (self[, axes])	Ignore labels from axes (replace those axes by “wildcard” axes).
<i>indexofmax</i> (self[, axis])	Returns indices of the maximum values along a given axis.
<i>indexofmin</i> (self[, axis])	Returns indices of the minimum values along a given axis.
<i>indicesofsorted</i> (self[, axis, ascending, kind])	Returns the indices that would sort this array.
<i>insert</i> (self, value[, before, after, pos, ...])	Inserts value in array along an axis.
<i>isin</i> (self, test_values[, assume_unique, invert])	Computes whether each element of this array is in <i>test_values</i> .
<i>items</i> (self[, axes, ascending])	Returns a (label, value) view of the array along axes.
<i>keys</i> (self[, axes, ascending])	Returns a view on the array labels along axes.
<i>labelofmax</i> (self[, axis])	Returns labels of the maximum values along a given axis.
<i>labelofmin</i> (self[, axis])	Returns labels of the minimum values along a given axis.
<i>labelsofsorted</i> (self[, axis, ascending, kind])	Returns the labels that would sort this array.
<i>max</i> (*axes_and_groups[, out, skipna, keepaxes])	Get maximum of array elements along given axes/groups.

Continued on next page

Table 28 – continued from previous page

<code>max_by(*axes_and_groups[, out, skipna, keep-axes])</code>	Get maximum of array elements for the given axes/groups.
<code>mean(*axes_and_groups[, dtype, out, skipna, ...])</code>	Computes the arithmetic mean.
<code>mean_by(*axes_and_groups[, dtype, out, ...])</code>	Computes the arithmetic mean.
<code>median(*axes_and_groups[, out, skipna, keep-axes])</code>	Computes the arithmetic median.
<code>median_by(*axes_and_groups[, out, skipna, ...])</code>	Computes the arithmetic median.
<code>min(*axes_and_groups[, out, skipna, keepaxes])</code>	Get minimum of array elements along given axes/groups.
<code>min_by(*axes_and_groups[, out, skipna, keep-axes])</code>	Get minimum of array elements for the given axes/groups.
<code>nonzero(self)</code>	Returns the indices of the elements that are non-zero.
<code>percent(self, *axes)</code>	Returns an array with values given as percent of the total of all values along given axes.
<code>percentile(q, *axes_and_groups[, out, ...])</code>	Computes the qth percentile of the data along the specified axis.
<code>percentile_by(q, *axes_and_groups[, out, ...])</code>	Computes the qth percentile of the data for the specified axis.
<code>posargmax(*args, **kwargs)</code>	
<code>posargmin(*args, **kwargs)</code>	
<code>posargsort(*args, **kwargs)</code>	
<code>prepend(self, axis, value[, label])</code>	Adds an array before self along an axis.
<code>prod(*axes_and_groups[, dtype, out, skipna, ...])</code>	Computes the product of array elements along given axes/groups.
<code>prod_by(*axes_and_groups[, dtype, out, ...])</code>	Computes the product of array elements for the given axes/groups.
<code>ptp(*axes_and_groups[, out])</code>	Returns the range of values (maximum - minimum).
<code>ratio(self, *axes)</code>	Returns an array with all values divided by the sum of values along given axes.
<code>rationot0(self, *axes)</code>	Returns an Array with values array / array.sum(axes) where the sum is not 0, 0 otherwise.
<code>reindex(self[, axes_to_reindex, new_axis, ...])</code>	Reorder and/or add new labels in axes.
<code>rename(self[, renames, to, inplace])</code>	Renames axes of the array.
<code>reshape(self, target_axes)</code>	Given a list of new axes, changes the shape of the array.
<code>reshape_like(self, target)</code>	Same as reshape but with an array as input.
<code>reverse(self[, axes])</code>	Reverse axes of an array
<code>roll(self[, axis, n])</code>	Rolls the cells of the array n-times to the right along axis.
<code>set(self, value, **kwargs)</code>	Sets a subset of array to value.
<code>set_axes(self[, axes_to_replace, new_axis, ...])</code>	Replace one, several or all axes of the array.
<code>set_labels(self[, axis, labels, inplace])</code>	Replaces the labels of one or several axes of the array.
<code>shift(self, axis[, n])</code>	Shifts the cells of the array n-times to the right along axis.
<code>sort_axes(self[, axes, ascending])</code>	Sorts axes of the array.
<code>sort_axis(*args, **kwargs)</code>	
<code>sort_values(self[, key, axis, ascending])</code>	Sorts values of the array.
<code>split_axes(self[, axes, sep, names, regex, ...])</code>	Split axes and returns a new array
<code>split_axis(*args, **kwargs)</code>	
<code>std(*axes_and_groups[, dtype, ddof, out, ...])</code>	Computes the sample standard deviation.

Continued on next page

Table 28 – continued from previous page

<code>std_by(*axes_and_groups[, dtype, ddof, out, ...])</code>	Computes the sample standard deviation.
<code>sum(*axes_and_groups[, dtype, out, skipna, ...])</code>	Computes the sum of array elements along given axes/groups.
<code>sum_by(*axes_and_groups[, dtype, out, ...])</code>	Computes the sum of array elements for the given axes/groups.
<code>to_clipboard(self, *args, **kwargs)</code>	Sends the content of the array to clipboard.
<code>to_csv(self, filepath[, sep, na_rep, wide, ...])</code>	Writes array to a csv file.
<code>to_excel(self[, filepath, sheet, position, ...])</code>	Writes array in the specified sheet of specified excel workbook.
<code>to_frame(self[, fold_last_axis_name, dropna])</code>	Converts an Array into a Pandas DataFrame.
<code>to_hdf(self, filepath, key)</code>	Writes array to a HDF file.
<code>to_series(self[, name, dropna])</code>	Converts an Array into a Pandas Series.
<code>to_stata(self, filepath_or_buffer, **kwargs)</code>	Writes array to a Stata .dta file.
<code>transpose(self, *args)</code>	Reorder axes.
<code>unique(self[, axes, sort, sep])</code>	Returns unique values (optionally along axes)
<code>values(self[, axes, ascending])</code>	Returns a view on the values of the array along axes.
<code>var(*axes_and_groups[, dtype, ddof, out, ...])</code>	Computes the unbiased variance.
<code>var_by(*axes_and_groups[, dtype, ddof, out, ...])</code>	Computes the unbiased variance.
<code>with_axes(*args, **kwargs)</code>	
<code>with_total(*args[, op, label])</code>	Add aggregated values (sum by default) along each axis.

Attributes

<code>T</code>	Reorder axes.
<code>axes</code>	
<code>data</code>	
<code>df</code>	Converts an Array into a Pandas DataFrame.
<code>dtype</code>	Returns the type of the data of the array.
<code>i</code>	Allows selection of a subset using indices of labels.
<code>iflat</code>	Access the array by index as if it was flat (one dimensional) and all its axes were combined.
<code>info</code>	Describes an Array (metadata + shape and labels for each axis).
<code>ipoints</code>	Allows selection of arbitrary items in the array based on their N-dimensional index.
<code>item</code>	
<code>memory_used</code>	Returns the memory consumed by the array in human readable form.
<code>meta</code>	Returns metadata of the array.
<code>nbytes</code>	Returns the number of bytes used to store the array in memory.
<code>ndim</code>	Returns the number of dimensions of the array.
<code>plot</code>	Plots the data of the array into a graph (window pop-up).
<code>points</code>	Allows selection of arbitrary items in the array based on their N-dimensional label index.
<code>series</code>	Converts an Array into a Pandas Series.
<code>shape</code>	Returns the shape of the array as a tuple.

Continued on next page

Table 29 – continued from previous page

<i>size</i>	Returns the number of elements in array.
<i>title</i>	

Array Creation Functions

<i>sequence</i> (axis[, initial, inc, mult, func, ...])	Creates an array by sequentially applying modifications to the array along axis.
<i>ndtest</i> (shape_or_axes[, start, label_start, ...])	Returns test array with given shape.
<i>zeros</i> (axes[, title, dtype, order, meta])	Returns an array with the specified axes and filled with zeros.
<i>zeros_like</i> (array[, title, dtype, order, meta])	Returns an array with the same axes as array and filled with zeros.
<i>ones</i> (axes[, title, dtype, order, meta])	Returns an array with the specified axes and filled with ones.
<i>ones_like</i> (array[, title, dtype, order, meta])	Returns an array with the same axes as array and filled with ones.
<i>empty</i> (axes[, title, dtype, order, meta])	Returns an array with the specified axes and uninitialized (arbitrary) data.
<i>empty_like</i> (array[, title, dtype, order, meta])	Returns an array with the same axes as array and uninitialized (arbitrary) data.
<i>full</i> (axes, fill_value[, title, dtype, ...])	Returns an array with the specified axes and filled with fill_value.
<i>full_like</i> (array, fill_value[, title, dtype, ...])	Returns an array with the same axes and type as input array and filled with fill_value.

larray.sequence

`larray.sequence` (*axis*, *initial*=0, *inc*=None, *mult*=1, *func*=None, *axes*=None, *title*=None, *meta*=None)
Creates an array by sequentially applying modifications to the array along axis.

The value for each label in axis will be given by sequentially transforming the value for the previous label. This transformation on the previous label value consists of applying the function “func” on that value if provided, or to multiply it by mult and increment it by inc otherwise.

Parameters

axis [axis definition (Axis, str, int)] Axis along which to apply mod. An axis definition can be passed as a string. An int will be interpreted as the length for a new anonymous axis.

initial [scalar or Array, optional] Value for the first label of axis. Defaults to 0.

inc [scalar, Array, optional] Value to increment the previous value by. Defaults to 0 if mult is provided, 1 otherwise.

mult [scalar, Array, optional] Value to multiply the previous value by. Defaults to 1.

func [function/callable, optional] Function to apply to the previous value. Defaults to None. Note that this is much slower than using inc and/or mult.

axes [int, tuple of int or tuple/list/AxisCollection of Axis, optional] Axes of the result. Defaults to the union of axes present in other arguments.

title [str, optional] Deprecated. See ‘meta’ below.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Examples

```
>>> year = Axis('year=2016..2019')
>>> sex = Axis('sex=M,F')
>>> sequence(year)
year 2016 2017 2018 2019
      0    1    2    3
>>> sequence('year=2016..2019')
year 2016 2017 2018 2019
      0    1    2    3
>>> sequence(year, 1.0, 0.5)
year 2016 2017 2018 2019
      1.0  1.5  2.0  2.5
>>> sequence(year, 1.0, mult=1.5)
year 2016 2017 2018 2019
      1.0  1.5  2.25 3.375
>>> inc = Array([1, 2], [sex])
>>> inc
sex  M  F
     1  2
>>> sequence(year, 1.0, inc)
sex\year 2016 2017 2018 2019
        M  1.0  2.0  3.0  4.0
        F  1.0  3.0  5.0  7.0
>>> mult = Array([2, 3], [sex])
>>> mult
sex  M  F
     2  3
>>> sequence(year, 1.0, mult=mult)
sex\year 2016 2017 2018 2019
        M  1.0  2.0  4.0  8.0
        F  1.0  3.0  9.0 27.0
>>> initial = Array([3, 4], [sex])
>>> initial
sex  M  F
     3  4
>>> sequence(year, initial, 1)
sex\year 2016 2017 2018 2019
        M   3   4   5   6
        F   4   5   6   7
>>> sequence(year, initial, mult=2)
sex\year 2016 2017 2018 2019
        M   3   6  12  24
        F   4   8  16  32
>>> sequence(year, initial, inc, mult)
sex\year 2016 2017 2018 2019
        M   3   7  15  31
        F   4  14  44 134
>>> def modify(prev_value):
...     return prev_value / 2
>>> sequence(year, 8, func=modify)
year 2016 2017 2018 2019
```

(continues on next page)

(continued from previous page)

```

      8      4      2      1
>>> sequence(3)
{0}*  0  1  2
      0  1  2
>>> sequence('year', axes=(sex, year))
sex\year  2016  2017  2018  2019
      M      0      1      2      3
      F      0      1      2      3

```

sequence can be used as the inverse of growth_rate:

```

>>> a = Array([1.0, 2.0, 3.0, 3.0], year)
>>> a
year  2016  2017  2018  2019
      1.0  2.0  3.0  3.0
>>> g = a.growth_rate() + 1
>>> g
year  2017  2018  2019
      2.0  1.5  1.0
>>> sequence(year, a[2016], mult=g)
year  2016  2017  2018  2019
      1.0  2.0  3.0  3.0

```

larray.ndtest

`larray.ndtest` (*shape_or_axes*, *start*=0, *label_start*=0, *title*=None, *dtype*=<class 'int'>, *meta*=None)

Returns test array with given shape.

Axes are named by single letters starting from 'a'. Axes labels are constructed using a '{axis_name}{label_pos}' pattern (e.g. 'a0'). Values start from *start* increase by steps of 1.

Parameters

shape_or_axes [int, tuple/list of int, str, single axis or tuple/list/AxisCollection of axes] If int or tuple/list of int, represents the shape of the array to create. In that case, default axes are generated. If string, it is used to generate axes (see [AxisCollection](#) constructor).

start [int or float, optional] Start value

label_start [int, optional] Label index for each axis is *label_start* + *position*. *label_start* defaults to 0.

title [str, optional] Deprecated. See 'meta' below.

dtype [type or np.dtype, optional] Type of resulting array.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

Create test array by passing a shape

```
>>> ndtest(6)
a  a0  a1  a2  a3  a4  a5
   0   1   2   3   4   5
>>> ndtest((2, 3))
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> ndtest((2, 3), label_start=1)
a\b  b1  b2  b3
a1   0   1   2
a2   3   4   5
>>> ndtest((2, 3), start=2)
a\b  b0  b1  b2
a0   2   3   4
a1   5   6   7
>>> ndtest((2, 3), dtype=float)
a\b  b0  b1  b2
a0  0.0  1.0  2.0
a1  3.0  4.0  5.0
```

Create test array by passing axes

```
>>> ndtest("nat=BE,FO;sex=M,F")
nat\sex  M  F
        BE  0  1
        FO  2  3
>>> nat = Axis("nat=BE,FO")
>>> sex = Axis("sex=M,F")
>>> ndtest([nat, sex])
nat\sex  M  F
        BE  0  1
        FO  2  3
```

larray.zeros

`larray.zeros` (*axes*, *title=None*, *dtype=<class 'float'>*, *order='C'*, *meta=None*)

Returns an array with the specified axes and filled with zeros.

Parameters

axes [int, tuple of int, Axis or tuple/list/AxisCollection of Axis] Collection of axes or a shape.

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

order [{‘C’, ‘F’}, optional] Whether to store multidimensional data in C- (default) or Fortran-contiguous (row- or column-wise) order in memory.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> zeros('nat=BE,FO;sex=M,F')
nat\sex    M    F
    BE  0.0  0.0
    FO  0.0  0.0
>>> zeros([(['BE', 'FO'], 'nat'),
...        ([['M', 'F'], 'sex'])])
nat\sex    M    F
    BE  0.0  0.0
    FO  0.0  0.0
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> zeros([nat, sex])
nat\sex    M    F
    BE  0.0  0.0
    FO  0.0  0.0
```

larray.zeros_like

`larray.zeros_like(array, title=None, dtype=None, order='K', meta=None)`

Returns an array with the same axes as `array` and filled with zeros.

Parameters

array [Array] Input array.

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Overrides the data type of the result.

order [{‘C’, ‘F’, ‘A’, or ‘K’}, optional] Overrides the memory layout of the result. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if *a* is Fortran contiguous, ‘C’ otherwise. ‘K’ (default) means match the layout of *a* as closely as possible.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> a = ndtest((2, 3))
>>> zeros_like(a)
a\b  b0  b1  b2
a0   0   0   0
a1   0   0   0
```

larray.ones

`larray.ones` (*axes*, *title=None*, *dtype=<class 'float'>*, *order='C'*, *meta=None*)

Returns an array with the specified axes and filled with ones.

Parameters

axes [int, tuple of int, Axis or tuple/list/AxisCollection of Axis] Collection of axes or a shape.

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Desired data-type for the array, e.g., *numpy.int8*. Default is *numpy.float64*.

order [{‘C’, ‘F’}, optional] Whether to store multidimensional data in C- (default) or Fortran-contiguous (row- or column-wise) order in memory.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```

>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> ones([nat, sex])
nat\sex    M    F
    BE  1.0  1.0
    FO  1.0  1.0

```

larray.ones_like

`larray.ones_like` (*array*, *title=None*, *dtype=None*, *order='K'*, *meta=None*)

Returns an array with the same axes as *array* and filled with ones.

Parameters

array [Array] Input array.

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Overrides the data type of the result.

order [{‘C’, ‘F’, ‘A’, or ‘K’}, optional] Overrides the memory layout of the result. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if *a* is Fortran contiguous, ‘C’ otherwise. ‘K’ (default) means match the layout of *a* as closely as possible.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> a = ndtest((2, 3))
>>> ones_like(a)
a\b  b0  b1  b2
a0    1   1   1
a1    1   1   1
```

larray.empty

`larray.empty` (*axes*, *title=None*, *dtype=<class 'float'>*, *order='C'*, *meta=None*)

Returns an array with the specified axes and uninitialized (arbitrary) data.

Parameters

axes [int, tuple of int, Axis or tuple/list/AxisCollection of Axis] Collection of axes or a shape.

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Desired data-type for the array, e.g., *numpy.int8*. Default is *numpy.float64*.

order [{‘C’, ‘F’}, optional] Whether to store multidimensional data in C- (default) or Fortran-contiguous (row- or column-wise) order in memory.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> empty([nat, sex])
nat\sex          M          F
   BE  2.47311483356e-315  2.47498446195e-315
   FO           0.0    6.07684618082e-31
```

larray.empty_like

`larray.empty_like` (*array*, *title=None*, *dtype=None*, *order='K'*, *meta=None*)

Returns an array with the same axes as array and uninitialized (arbitrary) data.

Parameters

array [Array] Input array.

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Overrides the data type of the result. Defaults to the data type of array.

order [{‘C’, ‘F’, ‘A’, or ‘K’}, optional] Overrides the memory layout of the result. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if *a* is Fortran contiguous, ‘C’ otherwise. ‘K’ (default) means match the layout of *a* as closely as possible.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> a = ndtest((3, 2))
>>> empty_like(a)
a\b          b0          b1
a0  2.12199579097e-314  6.36598737388e-314
a1  1.06099789568e-313  1.48539705397e-313
a2  1.90979621226e-313  2.33419537056e-313
```

larray.full

`larray.full` (*axes*, *fill_value*, *title=None*, *dtype=None*, *order='C'*, *meta=None*)

Returns an array with the specified axes and filled with *fill_value*.

Parameters

axes [int, tuple of int, Axis or tuple/list/AxisCollection of Axis] Collection of axes or a shape.

fill_value [scalar or Array] Value to fill the array

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Desired data-type for the array. Default is the data type of *fill_value*.

order [{‘C’, ‘F’}, optional] Whether to store multidimensional data in C- (default) or Fortran-contiguous (row- or column-wise) order in memory.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> full([nat, sex], 42.0)
nat\sex      M      F
      BE  42.0  42.0
      FO  42.0  42.0
>>> initial_value = ndtest([sex])
>>> initial_value
```

(continues on next page)

(continued from previous page)

```

sex  M  F
    0  1
>>> full([nat, sex], initial_value)
nat\sex  M  F
        BE 0  1
        FO 0  1

```

larray.full_like

`larray.full_like(array, fill_value, title=None, dtype=None, order='K', meta=None)`

Returns an array with the same axes and type as input array and filled with `fill_value`.

Parameters

array [Array] Input array.

fill_value [scalar or Array] Value to fill the array

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Overrides the data type of the result. Defaults to the data type of array.

order [{‘C’, ‘F’, ‘A’, or ‘K’}, optional] Overrides the memory layout of the result. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if *a* is Fortran contiguous, ‘C’ otherwise. ‘K’ (default) means match the layout of *a* as closely as possible.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```

>>> a = ndtest((2, 3))
>>> full_like(a, 5)
a\b  b0  b1  b2
a0    5   5   5
a1    5   5   5

```

Copying

<code>Array.copy(self)</code>	Returns a copy of the array.
<code>Array.astype(dtype[, order, casting, subok, ...])</code>	Copy of the array, cast to a specified type.

larray.Array.copy`Array.copy(self)`

Returns a copy of the array.

larray.Array.astype`Array.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)`

Copy of the array, cast to a specified type.

Parameters**dtype** [str or dtype] Typecode or data-type to which the array is cast.**order** [{ 'C', 'F', 'A', 'K' }, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.**casting** [{ 'no', 'equiv', 'safe', 'same_kind', 'unsafe' }, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok [bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.**copy** [bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the *dtype*, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.**Returns****arr_t** [ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with *dtype*, order given by *dtype*, *order*.**Raises****ComplexWarning** When casting from complex to float or int. To avoid this, one should use `a.real.astype(τ)`.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

Inspecting

Array.data	Data of the array (Numpy ndarray)
Array.axes	Axes of the array (AxisCollection)
Array.title	Title of the array (str)

<i>Array.info</i>	Describes an Array (metadata + shape and labels for each axis).
<i>Array.shape</i>	Returns the shape of the array as a tuple.
<i>Array.ndim</i>	Returns the number of dimensions of the array.
<i>Array.dtype</i>	Returns the type of the data of the array.
<i>Array.size</i>	Returns the number of elements in array.
<i>Array.nbytes</i>	Returns the number of bytes used to store the array in memory.
<i>Array.memory_used</i>	Returns the memory consumed by the array in human readable form.

larray.Array.info

property `Array.info`

Describes an Array (metadata + shape and labels for each axis).

Returns

str Description of the array (metadata + shape and labels for each axis).

Examples

```

>>> mat0 = Array([[2.0, 5.0], [8.0, 6.0]], "nat=BE,FO; sex=F,M")
>>> mat0.info
2 x 2
  nat [2]: 'BE' 'FO'
  sex [2]: 'F' 'M'
dtype: float64
memory used: 32 bytes
>>> mat0.meta.title = 'test matrix'
>>> mat0.info
title: test matrix
2 x 2
  nat [2]: 'BE' 'FO'
  sex [2]: 'F' 'M'
dtype: float64
memory used: 32 bytes

```

larray.Array.shape

property Array.shape

Returns the shape of the array as a tuple.

Returns

tuple Tuple representing the current shape.

Examples

```

>>> a = ndtest('nat=BE,FO;sex=M,F;type=type1,type2,type3')
>>> a.shape
(2, 2, 3)

```

larray.Array.ndim

property Array.ndim

Returns the number of dimensions of the array.

Returns

int Number of dimensions of an Array.

Examples

```

>>> a = ndtest('nat=BE,FO;sex=M,F')
>>> a.ndim
2

```

`larray.Array.dtype`

property `Array.dtype`

Returns the type of the data of the array.

Returns

dtype Type of the data of the array.

Examples

```
>>> a = zeros('sex=M,F;type=type1,type2,type3')
>>> a.dtype
dtype('float64')
```

`larray.Array.size`

property `Array.size`

Returns the number of elements in array.

Returns

int Number of elements in array.

Examples

```
>>> a = ndtest('sex=M,F;type=type1,type2,type3')
>>> a.size
6
```

`larray.Array.nbytes`

property `Array.nbytes`

Returns the number of bytes used to store the array in memory.

Returns

int Number of bytes in array.

Examples

```
>>> a = ndtest('sex=M,F;type=type1,type2,type3', dtype=float)
>>> a.nbytes
48
```

larray.Array.memory_used**property** `Array.memory_used`

Returns the memory consumed by the array in human readable form.

Returns

str Memory used by the array.

Examples

```
>>> a = ndtest('sex=M,F;type=type1,type2,type3', dtype=float)
>>> a.memory_used
'48 bytes'
```

Modifying/Selecting

<code>Array.i</code>	Allows selection of a subset using indices of labels.
<code>Array.points</code>	Allows selection of arbitrary items in the array based on their N-dimensional label index.
<code>Array.ipoints</code>	Allows selection of arbitrary items in the array based on their N-dimensional index.
<code>Array.iflat</code>	Access the array by index as if it was flat (one dimensional) and all its axes were combined.
<code>Array.set(self, value, **kwargs)</code>	Sets a subset of array to value.
<code>Array.drop(self[, labels])</code>	Return array without some labels or indices along an axis.
<code>Array.ignore_labels(self[, axes])</code>	Ignore labels from axes (replace those axes by “wild-card” axes).
<code>Array.filter(self[, collapse])</code>	Filters the array along the axes given as keyword arguments.
<code>Array.apply(self, transform, *args, **kwargs)</code>	Apply a transformation function to array elements.
<code>Array.apply_map(self, mapping[, dtype])</code>	Apply a transformation mapping to array elements.

larray.Array.i**Array.i**

Allows selection of a subset using indices of labels.

Examples

```
>>> arr = ndtest((2, 3, 4))
>>> arr
  a  b\c  c0  c1  c2  c3
a0  b0   0   1   2   3
a0  b1   4   5   6   7
a0  b2   8   9  10  11
a1  b0  12  13  14  15
a1  b1  16  17  18  19
a1  b2  20  21  22  23
```

```
>>> arr.i[:, 0:2, [0, 2]]
a  b\c  c0  c2
a0  b0   0   2
a0  b1   4   6
a1  b0  12  14
a1  b1  16  18
```

larray.Array.points

Array.points

Allows selection of arbitrary items in the array based on their N-dimensional label index.

Examples

```
>>> arr = ndtest((2, 3, 4))
>>> arr
a  b\c  c0  c1  c2  c3
a0  b0   0   1   2   3
a0  b1   4   5   6   7
a0  b2   8   9  10  11
a1  b0  12  13  14  15
a1  b1  16  17  18  19
a1  b2  20  21  22  23
```

To select the two points with label coordinates [a0, b0, c0] and [a1, b2, c2], you must do:

```
>>> arr.points(['a0', 'a1'], ['b0', 'b2'], ['c0', 'c2'])
a_b_c  a0_b0_c0  a1_b2_c2
          0          22
>>> arr.points['a0,a1', 'b0,b2', 'c0,c2']
a_b_c  a0_b0_c0  a1_b2_c2
          0          22
```

The number of label(s) on each dimension must be equal:

```
>>> arr.points['a0,a1', 'b0,b2', 'c0,c1,c2']
Traceback (most recent call last):
...
IndexError: shape mismatch: indexing arrays could not be broadcast together with
↳ shapes (2,) (2,) (3,)
```

larray.Array.ipoints

Array.ipoints

Allows selection of arbitrary items in the array based on their N-dimensional index.

Examples

```
>>> arr = ndtest((2, 3, 4))
>>> arr
a  b\c  c0  c1  c2  c3
a0  b0   0   1   2   3
a0  b1   4   5   6   7
a0  b2   8   9  10  11
a1  b0  12  13  14  15
a1  b1  16  17  18  19
a1  b2  20  21  22  23
```

To select the two points with index coordinates [0, 0, 0] and [1, 2, 2], you must do:

```
>>> arr.ipoints[[0,1], [0,2], [0,2]]
a_b_c  a0_b0_c0  a1_b2_c2
          0          22
```

The number of index(es) on each dimension must be equal:

```
>>> arr.ipoints[[0,1], [0,2], [0,1,2]]
Traceback (most recent call last):
...
IndexError: shape mismatch: indexing arrays could not be broadcast together with_
↳ shapes (2,) (2,) (3,)
```

larray.Array.iflat

Array.iflat

Access the array by index as if it was flat (one dimensional) and all its axes were combined.

Notes

In general `arr.iflat[key]` should be equivalent to (but much faster than) `arr.combine_axes().i[key]`

Examples

```
>>> arr = ndtest((2, 3)) * 10
>>> arr
a\b  b0  b1  b2
a0   0  10  20
a1  30  40  50
```

To select the first, second, fourth and fifth values across all axes:

```
>>> arr.combine_axes().i[[0, 1, 3, 4]]
a_b  a0_b0  a0_b1  a1_b0  a1_b1
      0     10     30     40
>>> arr.iflat[[0, 1, 3, 4]]
a_b  a0_b0  a0_b1  a1_b0  a1_b1
      0     10     30     40
```

Set the first and sixth values to 42

```
>>> arr.iflat[[0, 5]] = 42
>>> arr
a\b  b0  b1  b2
a0   42  10  20
a1   30  40  42
```

When the key is an Array, the result will have the axes of the key

```
>>> key = Array([0, 3], 'c=c0,c1')
>>> key
c  c0  c1
   0   3
>>> arr.iflat[key]
c  c0  c1
   42  30
```

larray.Array.set

`Array.set(self, value, **kwargs)`

Sets a subset of array to value.

- all common axes must be either of length 1 or the same length
- extra axes in value must be of length 1
- extra axes in current array can have any length

Parameters

value [scalar or Array]

Examples

```
>>> arr = ndtest((3, 3))
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
a2   6   7   8
>>> arr['a1:', 'b1:'].set(10)
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3  10  10
a2   6  10  10
>>> arr['a1:', 'b1:'].set(ndtest("a=a1,a2;b=b1,b2"))
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3   0   1
a2   6   2   3
```

larray.Array.drop

`Array.drop(self, labels=None)`

Return array without some labels or indices along an axis.

Parameters

labels [scalar, list or Group] Label(s) or group to remove. To remove indices, one must pass an IGroup.

Returns

Array Array with *labels* removed along their axis.

Examples

```
>>> arr1 = ndtest((2, 4))
>>> arr1
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
>>> a, b = arr1.axes
```

dropping a single label

```
>>> arr1.drop('b1')
a\b  b0  b2  b3
a0    0   2   3
a1    4   6   7
```

dropping multiple labels

```
>>> # arr1.drop('b1,b3')
>>> arr1.drop(['b1', 'b3'])
a\b  b0  b2
a0    0   2
a1    4   6
```

dropping a slice

```
>>> # arr1.drop('b1:b3')
>>> arr1.drop(b['b1':'b3'])
a\b  b0
a0    0
a1    4
```

when deleting indices instead of labels, one must specify the axis explicitly (using an IGroup):

```
>>> # arr1.drop('b.i[1]')
>>> arr1.drop(b.i[1])
a\b  b0  b2  b3
a0    0   2   3
a1    4   6   7
```

as when deleting ambiguous labels (which are present on several axes):

```
>>> a = Axis('a=label0..label2')
>>> b = Axis('b=label0..label2')
>>> arr2 = ndtest((a, b))
>>> arr2
  a\b  label0  label1  label2
label0      0      1      2
label1      3      4      5
label2      6      7      8
>>> # arr2.drop('a[label1]')
>>> arr2.drop(a['label1'])
  a\b  label0  label1  label2
label0      0      1      2
label2      6      7      8
```

larray.Array.ignore_labels

Array.**ignore_labels**(*self*, *axes=None*)

Ignore labels from axes (replace those axes by “wildcard” axes).

Useful when you want to apply operations between two arrays or subarrays with same shape but incompatible axes (different labels).

Parameters

axes [Axis or list/tuple/AxisCollection of Axis, optional] Axis(es) on which you want to drop the labels.

Returns

Array

Notes

Use it at your own risk.

Examples

```
>>> a = Axis('a=a1,a2')
>>> b = Axis('b=b1,b2')
>>> b2 = Axis('b=b2,b3')
>>> arr1 = ndtest([a, b])
>>> arr1
a\b  b1  b2
a1   0   1
a2   2   3
>>> arr1.ignore_labels(b)
a\b*  0   1
a1   0   1
a2   2   3
>>> arr1.ignore_labels([a, b])
a*\b*  0   1
      0   0   1
      1   2   3
>>> arr2 = ndtest([a, b2])
```

(continues on next page)

(continued from previous page)

```

>>> arr2
a\b  b2  b3
a1    0   1
a2    2   3
>>> arr1 * arr2
Traceback (most recent call last):
...
ValueError: incompatible axes:
Axis(['b2', 'b3'], 'b')
vs
Axis(['b1', 'b2'], 'b')
>>> arr1 * arr2.ignore_labels()
a\b  b1  b2
a1    0   1
a2    4   9
>>> arr1.ignore_labels() * arr2
a\b  b2  b3
a1    0   1
a2    4   9
>>> arr1.ignore_labels('a') * arr2.ignore_labels('b')
a\b  b1  b2
a1    0   1
a2    4   9

```

larray.Array.filter

Array.**filter** (*self*, *collapse=False*, ***kwargs*)

Filters the array along the axes given as keyword arguments.

The *collapse* argument determines whether consecutive ranges should be collapsed to slices, which is more efficient and returns a view (and not a copy) if possible (if all ranges are consecutive). Only use this argument if you do not intent to modify the resulting array, or if you know what you are doing.

It is similar to `np.take` but works with several axes at once.

larray.Array.apply

Array.**apply** (*self*, *transform*, **args*, ***kwargs*)

Apply a transformation function to array elements.

Parameters

transform [function] Function to apply. This function will be called in turn with each element of the array as the first argument and must return an Array, scalar or tuple. If returning arrays the axes of those arrays must be the same for all calls to the function.

***args** Extra arguments to pass to the function.

by [str, int or Axis or tuple/list/AxisCollection of the them, optional] Axis or axes along which to iterate. The function will thus be called with arrays having all axes not mentioned. Defaults to None (all axes). Mutually exclusive with the *axes* argument.

axes [str, int or Axis or tuple/list/AxisCollection of the them, optional] Axis or axes the arrays passed to the function will have. Defaults to None (the function is given scalars). Mutually exclusive with the *by* argument.

dtype [type or list of types, optional] Output(s) data type(s). Defaults to None (inspect all output values to infer it automatically).

ascending [bool, optional] Whether or not to iterate the axes in ascending order (from start to end). Defaults to True.

****kwargs** Extra keyword arguments are passed to the function (as keyword arguments).

Returns

Array or scalar, or tuple of them Axes will be the union of those in axis and those of values returned by the function.

Examples

First let us define a test array

```
>>> arr = Array([[0, 2, 1],
...              [3, 1, 5]], 'a=a0,a1;b=b0..b2')
>>> arr
a\b  b0  b1  b2
a0    0   2   1
a1    3   1   5
```

Here is a simple function we would like to apply to each element of the array. Note that this particular example should rather be written as: `arr ** 2` as it is both more concise and much faster.

```
>>> def square(x):
...     return x ** 2
>>> arr.apply(square)
a\b  b0  b1  b2
a0    0   4   1
a1    9   1  25
```

Functions can also be applied along some axes:

```
>>> # this is equivalent to (but much slower than): arr.sum('a')
... arr.apply(sum, axes='a')
b  b0  b1  b2
   3   3   6
>>> # this is equivalent to (but much slower than): arr.sum_by('a')
... arr.apply(sum, by='a')
a  a0  a1
   3   9
```

Applying the function along some axes will return an array with the union of those axes and the axes of the returned values. For example, let us define a function which returns the k highest values of an array.

```
>>> def topk(a, k=2):
...     return a.sort_values(ascending=False).ignore_labels().i[:k]
>>> arr.apply(topk, by='a')
a\b*  0  1
a0    2  1
a1    5  3
```

Other arguments can be passed to the function:

```
>>> arr.apply(topk, 3, by='a')
a\b*  0  1  2
a0    2  1  0
a1    5  3  1
```

or by using keyword arguments:

```
>>> arr.apply(topk, by='a', k=3)
a\b*  0  1  2
a0    2  1  0
a1    5  3  1
```

If the function returns several values (as a tuple), the result will be a tuple of arrays. For example, let use define a function which decompose an array in its mean and the difference to that mean :

```
>>> def mean_decompose(a):
...     mean = a.mean()
...     return mean, a - mean
>>> mean_by_a, diff_to_mean = arr.apply(mean_decompose, by='a')
>>> mean_by_a
a   a0   a1
   1.0  3.0
>>> diff_to_mean
a\b   b0   b1   b2
a0  -1.0  1.0  0.0
a1   0.0 -2.0  2.0
```

larray.Array.apply_map

`Array.apply_map(self, mapping, dtype=None)`

Apply a transformation mapping to array elements.

Parameters

mapping [mapping (dict)] Mapping to apply to values of the array. A mapping (dict) must have the values to transform as keys and the new values as values, that is: {<oldvalue1>: <newvalue1>, <oldvalue2>: <newvalue2>, ...}.

dtype [type, optional] Output dtype. Defaults to None (inspect all output values to infer it automatically).

Returns

Array Axes will be the same as the original array axes.

Notes

To apply a transformation given as an Array (with current values as labels on one axis of the array and desired values as the array values), you can use: `mapping_arr[original_arr]`.

Examples

First let us define a test array

```
>>> arr = Array([[0, 2, 1],
...              [3, 1, 5]], 'a=a0,a1;b=b0..b2')
>>> arr
a\b  b0  b1  b2
a0    0   2   1
a1    3   1   5
```

Now, assuming for a moment that the values of our test array above were in fact some numeric representation of names and we had the correspondence to the actual names stored in a dictionary:

```
>>> code_to_names = {0: 'foo', 1: 'bar', 2: 'baz',
...                  3: 'boo', 4: 'far', 5: 'faz'}
```

We could get back an array with the actual names by using:

```
>>> arr.apply_map(code_to_names)
a\b  b0  b1  b2
a0  foo baz bar
a1  boo bar faz
```

Changing Axes or Labels

<code>Array.set_axes(self[, axes_to_replace, ...])</code>	Replace one, several or all axes of the array.
<code>Array.rename(self[, renames, to, inplace])</code>	Renames axes of the array.
<code>Array.set_labels(self[, axis, labels, inplace])</code>	Replaces the labels of one or several axes of the array.
<code>Array.combine_axes(self[, axes, sep, wildcard])</code>	Combine several axes into one.
<code>Array.split_axes(self[, axes, sep, names, ...])</code>	Split axes and returns a new array
<code>Array.reverse(self[, axes])</code>	Reverse axes of an array

larray.Array.set_axes

`Array.set_axes` (*self*, *axes_to_replace*=None, *new_axis*=None, *inplace*=False, ***kwargs*)

Replace one, several or all axes of the array.

Parameters

axes_to_replace [axis ref or dict {axis ref: axis} or list of tuple (axis ref, axis)]
or list of Axis or AxisCollection

Axes to replace. If a single axis reference is given, the *new_axis* argument must be provided. If a list of Axis or an AxisCollection is given, all axes will be replaced by the new ones. In that case, the number of new axes must match the number of the old ones.

new_axis [Axis, optional] New axis if *axes_to_replace* contains a single axis reference.

inplace [bool, optional] Whether or not to modify the original object or return a new array and leave the original intact. Defaults to False.

****kwargs** [Axis] New axis for each axis to replace given as a keyword argument.

Returns

Array Array with axes replaced.

See also:

`rename` rename one of several axes

Examples

```
>>> arr = ndtest((2, 3))
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> row = Axis(['r0', 'r1'], 'row')
>>> column = Axis(['c0', 'c1', 'c2'], 'column')
```

Replace one axis (second argument *new_axis* must be provided)

```
>>> arr.set_axes('a', row)
row\b  b0  b1  b2
r0     0   1   2
r1     3   4   5
```

Replace several axes (keywords, list of tuple or dictionary)

```
>>> arr.set_axes(a=row, b=column)
>>> # or
>>> arr.set_axes([('a', row), ('b', column)])
>>> # or
>>> arr.set_axes({'a': row, 'b': column})
row\column c0  c1  c2
r0         0   1   2
r1         3   4   5
```

Replace all axes (list of axes or AxisCollection)

```
>>> arr.set_axes([row, column])
row\column c0  c1  c2
r0         0   1   2
r1         3   4   5
>>> arr2 = ndtest([row, column])
>>> arr.set_axes(arr2.axes)
row\column c0  c1  c2
r0         0   1   2
r1         3   4   5
```

`larray.Array.rename`

`Array.rename` (*self*, *renames=None*, *to=None*, *inplace=False*, ***kwargs*)

Renames axes of the array.

Parameters

renames [axis ref or dict {axis ref: str} or list of tuple (axis ref, str)] Renames to apply. If a single axis reference is given, the *to* argument must be used.

to [str or Axis] New name if *renames* contains a single axis reference.

****kwargs** [str or Axis] New name for each axis given as a keyword argument.

Returns

Array Array with axes renamed.

See also:

[`set_axes`](#) replace one or several axes

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> arr = ndtest([nat, sex])
>>> arr
nat\sex  M  F
      BE  0  1
      FO  2  3
>>> arr.rename(nat, 'nat2')
nat2\sex  M  F
      BE  0  1
      FO  2  3
>>> arr.rename(nat='nat2', sex='sex2')
nat2\sex2  M  F
      BE  0  1
      FO  2  3
>>> arr.rename([('nat', 'nat2'), ('sex', 'sex2')])
nat2\sex2  M  F
      BE  0  1
      FO  2  3
>>> arr.rename({'nat': 'nat2', 'sex': 'sex2'})
nat2\sex2  M  F
      BE  0  1
      FO  2  3
```

larray.Array.set_labels

`Array.set_labels(self, axis=None, labels=None, inplace=False, **kwargs)`

Replaces the labels of one or several axes of the array.

Parameters

axis [string or Axis or dict] Axis for which we want to replace labels, or mapping {axis: changes} where changes can either be the complete list of labels, a mapping {old_label: new_label} or a function to transform labels. If there is no ambiguity (two or more axes have the same labels), *axis* can be a direct mapping {old_label: new_label}.

labels [int, str, iterable or mapping or function, optional] Integer or list of values usable as the collection of labels for an Axis. If this is mapping, it must be {old_label: new_label}. If it is a function, it must be a function accepting a single argument (a label) and returning a single value. This argument must not be used if *axis* is a mapping.

inplace [bool, optional] Whether or not to modify the original object or return a new array and leave the original intact. Defaults to False.

****kwargs** : *axis*='labels for each axis you want to set labels.

Returns

Array Array with modified labels.

See also:

[*AxisCollection.set_labels*](#)

Examples

```
>>> a = ndtest('nat=BE,FO;sex=M,F')
>>> a
nat\sex  M  F
      BE  0  1
      FO  2  3
>>> a.set_labels('sex', ['Men', 'Women'])
nat\sex  Men  Women
      BE    0    1
      FO    2    3
```

when passing a single string as labels, it will be interpreted to create the list of labels, so that one can use the same syntax than during axis creation.

```
>>> a.set_labels('sex', 'Men,Women')
nat\sex  Men  Women
      BE    0    1
      FO    2    3
```

to replace only some labels, one must give a mapping giving the new label for each label to replace

```
>>> a.set_labels('sex', {'M': 'Men'})
nat\sex  Men  F
      BE    0  1
      FO    2  3
```

to transform labels by a function, use any function accepting and returning a single argument:

```
>>> a.set_labels('nat', str.lower)
nat\sex  M  F
      be  0  1
      fo  2  3
```

to replace labels for several axes at the same time, one should give a mapping giving the new labels for each changed axis

```
>>> a.set_labels({'sex': 'Men,Women', 'nat': 'Belgian,Foreigner'})
nat\sex  Men  Women
Belgian    0    1
Foreigner  2    3
```

or use keyword arguments

```
>>> a.set_labels(sex='Men,Women', nat='Belgian,Foreigner')
nat\sex  Men  Women
Belgian    0    1
Foreigner  2    3
```

one can also replace some labels in several axes by giving a mapping of mappings

```
>>> a.set_labels({'sex': {'M': 'Men'}, 'nat': {'BE': 'Belgian'}})
nat\sex  Men  F
Belgian    0  1
      FO    2  3
```

when there is no ambiguity (two or more axes have the same labels), it is possible to give a mapping between old and new labels

```
>>> a.set_labels({'M': 'Men', 'BE': 'Belgian'})
nat\sex  Men  F
Belgian    0  1
      FO    2  3
```

larray.Array.combine_axes

Array.**combine_axes** (*self*, *axes=None*, *sep='_'*, *wildcard=False*)

Combine several axes into one.

Parameters

axes [tuple, list, AxisCollection of axes or list of combination of those or dict, optional] axes to combine. Tuple, list or AxisCollection will combine several axes into one. To chain several axes combinations, pass a list of tuple/list/AxisCollection of axes. To set the name(s) of resulting axis(es), use a {(axes, to, combine): 'new_axis_name'} dictionary. Defaults to all axes.

sep [str, optional] delimiter to use for combining. Defaults to '_'.

wildcard [bool, optional] whether or not to produce a wildcard axis even if the axes to combine are not. This is much faster, but loose axes labels.

Returns

Array Array with combined axes.

Examples

```
>>> arr = ndtest((2, 3))
>>> arr
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
>>> arr.combine_axes()
a_b  a0_b0  a0_b1  a0_b2  a1_b0  a1_b1  a1_b2
      0      1      2      3      4      5
>>> arr.combine_axes(sep='/')
a/b  a0/b0  a0/b1  a0/b2  a1/b0  a1/b1  a1/b2
      0      1      2      3      4      5
>>> arr = ndtest((2, 2, 2, 2))
>>> arr
a  b  c\d  d0  d1
a0 b0  c0  0   1
a0 b0  c1  2   3
a0 b1  c0  4   5
a0 b1  c1  6   7
a1 b0  c0  8   9
a1 b0  c1 10  11
a1 b1  c0 12  13
a1 b1  c1 14  15
>>> arr.combine_axes (('a', 'c'))
a_c  b\d  d0  d1
a0_c0 b0  0   1
a0_c0 b1  4   5
a0_c1 b0  2   3
a0_c1 b1  6   7
a1_c0 b0  8   9
a1_c0 b1 12  13
a1_c1 b0 10  11
a1_c1 b1 14  15
>>> arr.combine_axes (('a', 'c'): 'ac')
ac  b\d  d0  d1
a0_c0 b0  0   1
a0_c0 b1  4   5
a0_c1 b0  2   3
a0_c1 b1  6   7
a1_c0 b0  8   9
a1_c0 b1 12  13
a1_c1 b0 10  11
a1_c1 b1 14  15
```

make several combinations at once

```
>>> arr.combine_axes (('a', 'c'), ('b', 'd'))
a_c\b_d  b0_d0  b0_d1  b1_d0  b1_d1
a0_c0    0      1      4      5
a0_c1    2      3      6      7
a1_c0    8      9     12     13
a1_c1   10     11     14     15
>>> arr.combine_axes (('a', 'c'): 'ac', ('b', 'd'): 'bd')
ac\b_d  b0_d0  b0_d1  b1_d0  b1_d1
a0_c0    0      1      4      5
a0_c1    2      3      6      7
```

(continues on next page)

(continued from previous page)

a1_c0	8	9	12	13
a1_c1	10	11	14	15

larray.Array.split_axes`Array.split_axes(self, axes=None, sep='_', names=None, regex=None, sort=False, fill_value=nan)`

Split axes and returns a new array

Parameters

axes [int, str, Axis or any combination of those] axes to split. All labels *must* contain the given delimiter string. To split several axes at once, pass a list or tuple of axes to split. To set the names of resulting axes, use a {'axis_to_split': (new, axes)} dictionary. Defaults to all axes whose name contains the *sep* delimiter.

sep [str, optional] delimiter to use for splitting. Defaults to '_'. When *regex* is provided, the delimiter is only used on *names* if given as one string or on axis name if *names* is None.

names [str or list of str, optional] names of resulting axes. Defaults to None.

regex [str, optional] use regex instead of delimiter to split labels. Defaults to None.

sort [bool, optional] Whether or not to sort the combined axis before splitting it. When all combinations of labels are present in the combined axis, sorting is faster than not sorting. Defaults to False.

fill_value [scalar or Array, optional] Value to use for missing values when the combined axis does not contain all combination of labels. Defaults to NaN.

Returns

Array

Examples

```
>>> arr = ndtest((2, 3))
>>> arr
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
>>> combined = arr.combine_axes()
>>> combined
a_b  a0_b0  a0_b1  a0_b2  a1_b0  a1_b1  a1_b2
      0      1      2      3      4      5
>>> combined.split_axes()
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
```

Split labels using regex

```
>>> combined = ndtest('a_b=a0b0..a1b2')
>>> combined
a_b  a0b0  a0b1  a0b2  a1b0  a1b1  a1b2
      0      1      2      3      4      5
>>> combined.split_axes('a_b', regex=r'(\w{2})(\w{2})')
```

(continues on next page)

(continued from previous page)

a\b	b0	b1	b2
a0	0	1	2
a1	3	4	5

Split several axes at once

```
>>> combined = ndtest('a_b=a0_b0..a1_b1; c_d=c0_d0..c1_d1')
>>> combined
a_b\c_d  c0_d0  c0_d1  c1_d0  c1_d1
a0_b0      0      1      2      3
a0_b1      4      5      6      7
a1_b0      8      9     10     11
a1_b1     12     13     14     15
>>> # equivalent to combined.split_axes() which split all axes whose name_
↳ contains the `sep` delimiter.
>>> combined.split_axes(['a_b', 'c_d'])
a  b  c\d  d0  d1
a0 b0  c0   0   1
a0 b0  c1   2   3
a0 b1  c0   4   5
a0 b1  c1   6   7
a1 b0  c0   8   9
a1 b0  c1  10  11
a1 b1  c0  12  13
a1 b1  c1  14  15
>>> combined.split_axes({'a_b': ('A', 'B'), 'c_d': ('C', 'D')})
A  B  C\D  d0  d1
a0 b0  c0   0   1
a0 b0  c1   2   3
a0 b1  c0   4   5
a0 b1  c1   6   7
a1 b0  c0   8   9
a1 b0  c1  10  11
a1 b1  c0  12  13
a1 b1  c1  14  15
```

larray.Array.reverseArray.**reverse** (*self*, *axes=None*)

Reverse axes of an array

Parameters

axes [int, str, Axis or any combination of those] axes to reverse. If None, all axes are reversed. Defaults to None.

Returns

Array Array with passed *axes* reversed.

Examples

```
>>> arr = ndtest((2, 2, 2))
>>> arr
  a  b\c  c0  c1
a0  b0   0   1
a0  b1   2   3
a1  b0   4   5
a1  b1   6   7
```

Reverse one axis

```
>>> arr.reverse('c')
  a  b\c  c1  c0
a0  b0   1   0
a0  b1   3   2
a1  b0   5   4
a1  b1   7   6
```

Reverse several axes

```
>>> arr.reverse(('a', 'c'))
  a  b\c  c1  c0
a1  b0   5   4
a1  b1   7   6
a0  b0   1   0
a0  b1   3   2
```

Reverse all axes

```
>>> arr.reverse()
  a  b\c  c1  c0
a1  b1   7   6
a1  b0   5   4
a0  b1   3   2
a0  b0   1   0
```

Aggregation Functions

<code>Array.sum(*axes_and_groups[, dtype, out, ...])</code>	Computes the sum of array elements along given axes/groups.
<code>Array.sum_by(*axes_and_groups[, dtype, out, ...])</code>	Computes the sum of array elements for the given axes/groups.
<code>Array.prod(*axes_and_groups[, dtype, out, ...])</code>	Computes the product of array elements along given axes/groups.
<code>Array.prod_by(*axes_and_groups[, dtype, ...])</code>	Computes the product of array elements for the given axes/groups.
<code>Array.cumsum(self[, axis])</code>	Returns the cumulative sum of array elements along an axis.
<code>Array.cumprod(self[, axis])</code>	Returns the cumulative product of array elements.
<code>Array.mean(*axes_and_groups[, dtype, out, ...])</code>	Computes the arithmetic mean.
<code>Array.mean_by(*axes_and_groups[, dtype, ...])</code>	Computes the arithmetic mean.
<code>Array.median(*axes_and_groups[, out, ...])</code>	Computes the arithmetic median.

Continued on next page

Table 35 – continued from previous page

<code>Array.median_by(*axes_and_groups[, out, ...])</code>	Computes the arithmetic median.
<code>Array.var(*axes_and_groups[, dtype, ddof, ...])</code>	Computes the unbiased variance.
<code>Array.var_by(*axes_and_groups[, dtype, ...])</code>	Computes the unbiased variance.
<code>Array.std(*axes_and_groups[, dtype, ddof, ...])</code>	Computes the sample standard deviation.
<code>Array.std_by(*axes_and_groups[, dtype, ...])</code>	Computes the sample standard deviation.
<code>Array.percentile(q, *axes_and_groups[, out, ...])</code>	Computes the qth percentile of the data along the specified axis.
<code>Array.percentile_by(q, *axes_and_groups[, ...])</code>	Computes the qth percentile of the data for the specified axis.
<code>Array.ptp(*axes_and_groups[, out])</code>	Returns the range of values (maximum - minimum).
<code>Array.with_total(*args[, op, label])</code>	Add aggregated values (sum by default) along each axis.
<code>Array.percent(self, *axes)</code>	Returns an array with values given as percent of the total of all values along given axes.
<code>Array.ratio(self, *axes)</code>	Returns an array with all values divided by the sum of values along given axes.
<code>Array.rationot0(self, *axes)</code>	Returns an Array with values array / array.sum(axes) where the sum is not 0, 0 otherwise.
<code>Array.growth_rate(self[, axis, d, label])</code>	Calculates the growth along a given axis.
<code>Array.describe(self, *args, **kwargs)</code>	Descriptive summary statistics, excluding NaN values.
<code>Array.describe_by(self, *args, **kwargs)</code>	Descriptive summary statistics, excluding NaN values, along axes or for groups.

larray.Array.sum

`Array.sum(*axes_and_groups, dtype=None, out=None, skipna=None, keepaxes=False, **explicit_axes)`
Computes the sum of array elements along given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the sum is performed. The default (no axis or group) is to perform the sum over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the sum over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.

- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[*Array.sum_by*](#), [*Array.prod*](#), [*Array.prod_by*](#)

[*Array.cumsum*](#), [*Array.cumprod*](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.sum()
120
>>> # along axis 'a'
>>> arr.sum('a')
b  b0  b1  b2  b3
   24  28  32  36
>>> # along axis 'b'
>>> arr.sum('b')
a  a0  a1  a2  a3
   6  22  38  54
```

Select some rows only

```
>>> arr.sum(['a0', 'a1'])
b  b0  b1  b2  b3
   4   6   8  10
>>> # or equivalently
>>> # arr.sum('a0,a1')
```

Split an axis in several parts

```
>>> arr.sum(['a0', 'a1'], ['a2', 'a3'])
a\b  b0  b1  b2  b3
a0,a1  4   6   8  10
a2,a3 20  22  24  26
>>> # or equivalently
>>> # arr.sum('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.sum(X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23')
a\b  b0  b1  b2  b3
a01  4   6   8  10
a23 20  22  24  26
>>> # or equivalently
>>> # arr.sum('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.sum_by

`Array.sum_by(*axes_and_groups, dtype=None, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the sum of array elements for the given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The sum is performed along all axes except the given one(s). For groups, sum is performed along groups and non associated axes. The default (no axis or group) is to perform the sum over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the sum over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- ('a1', 'a3', 'a5', 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. `keepaxes='summation'`). Defaults to False.

Returns

Array or scalar

See also:

[Array.sum](#), [Array.prod](#), [Array.prod_by](#)

[Array.cumsum](#), [Array.cumprod](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.sum_by()
120
>>> # along axis 'a'
>>> arr.sum_by('a')
a  a0  a1  a2  a3
   6  22  38  54
>>> # along axis 'b'
>>> arr.sum_by('b')
b  b0  b1  b2  b3
   24  28  32  36
```

Select some rows only

```
>>> arr.sum_by(['a0', 'a1'])
28
>>> # or equivalently
>>> # arr.sum_by('a0,a1')
```

Split an axis in several parts

```
>>> arr.sum_by(['a0', 'a1'], ['a2', 'a3'])
a  a0,a1  a2,a3
   28     92
>>> # or equivalently
>>> # arr.sum_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.sum_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01  a23
   28   92
>>> # or equivalently
>>> # arr.sum_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.prod

`Array.prod(*axes_and_groups, dtype=None, out=None, skipna=None, keepaxes=False, **explicit_axes)`
 Computes the product of array elements along given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the product is performed. The default (no axis or group) is to perform the product over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the product over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

Array.prod_by, *Array.sum*, *Array.sum_by*

Array.cumsum, *Array.cumprod*

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.prod()
0
>>> # along axis 'a'
>>> arr.prod('a')
b  b0  b1  b2  b3
   0 585 1680 3465
>>> # along axis 'b'
>>> arr.prod('b')
a  a0  a1  a2  a3
   0 840 7920 32760
```

Select some rows only

```
>>> arr.prod(['a0', 'a1'])
b  b0  b1  b2  b3
   0   5  12  21
>>> # or equivalently
>>> # arr.prod('a0,a1')
```

Split an axis in several parts

```
>>> arr.prod(['a0', 'a1'], ['a2', 'a3'])
a\b  b0  b1  b2  b3
a0,a1  0   5  12  21
a2,a3 96 117 140 165
>>> # or equivalently
>>> # arr.prod('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.prod((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b  b0  b1  b2  b3
a01  0   5  12  21
a23 96 117 140 165
>>> # or equivalently
>>> # arr.prod('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.prod_by

`Array.prod_by(*axes_and_groups, dtype=None, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the product of array elements for the given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The product is performed along all axes except the given one(s). For groups, product is performed along groups and non associated axes. The default (no axis or group) is to perform the product over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the product over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

Array.prod, Array.sum, Array.sum_by
Array.cumsum, Array.cumprod

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.prod_by()
0
>>> # along axis 'a'
>>> arr.prod_by('a')
a  a0  a1  a2  a3
   0 840 7920 32760
>>> # along axis 'b'
>>> arr.prod_by('b')
b  b0  b1  b2  b3
   0 585 1680 3465
```

Select some rows only

```
>>> arr.prod_by(['a0', 'a1'])
0
>>> # or equivalently
>>> # arr.prod_by('a0,a1')
```

Split an axis in several parts

```
>>> arr.prod_by(['a0', 'a1'], ['a2', 'a3'])
a  a0,a1      a2,a3
   0 259459200
>>> # or equivalently
>>> # arr.prod_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.prod_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01      a23
   0 259459200
>>> # or equivalently
>>> # arr.prod_by('a0,a1>>a01;a2,a3>>a23')
```


larray.Array.cumsum`Array.cumsum(self, axis=-1)`

Returns the cumulative sum of array elements along an axis.

Parameters

axis [int or str or Axis, optional] Axis along which to perform the cumulative sum. If given as position, it can be a negative integer, in which case it counts from the last to the first axis. By default, the cumulative sum is performed along the last axis.

Returns

Array or scalar

See also:

Array.cumprod, Array.sum, Array.sum_by

Array.prod, Array.prod_by

Notes

Cumulative aggregation functions accept only one axis

Examples

```

>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
a2    8   9  10  11
a3   12  13  14  15
>>> arr.cumsum()
a\b  b0  b1  b2  b3
a0    0   1   3   6
a1    4   9  15  22
a2    8  17  27  38
a3   12  25  39  54
>>> arr.cumsum('a')
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   6   8  10
a2   12  15  18  21
a3   24  28  32  36

```

`larray.Array.cumprod`

`Array.cumprod(self, axis=-1)`

Returns the cumulative product of array elements.

Parameters

axis [int or str or Axis, optional] Axis along which to perform the cumulative product. If given as position, it can be a negative integer, in which case it counts from the last to the first axis. By default, the cumulative product is performed along the last axis.

Returns

Array or scalar

See also:

[`Array.cumsum`](#), [`Array.sum`](#), [`Array.sum_by`](#)

[`Array.prod`](#), [`Array.prod_by`](#)

Notes

Cumulative aggregation functions accept only one axis.

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
a2    8   9  10  11
a3   12  13  14  15
>>> arr.cumprod()
a\b  b0  b1  b2  b3
a0    0   0   0   0
a1    4  20 120 840
a2    8  72 720 7920
a3   12 156 2184 32760
>>> arr.cumprod('a')
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    0   5  12  21
a2    0  45 120 231
a3    0 585 1680 3465
```

larray.Array.mean

`Array.mean(*axes_and_groups, dtype=None, out=None, skipna=None, keepaxes=False, **explicit_axes)`
 Computes the arithmetic mean.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the mean is performed. The default (no axis or group) is to perform the mean over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the mean over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[`Array.mean_by`](#), [`Array.median`](#), [`Array.median_by`](#)

[`Array.var`](#), [`Array.var_by`](#), [`Array.std`](#), [`Array.std_by`](#)

*Array.percentile, Array.percentile_by***Examples**

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.mean()
7.5
>>> # along axis 'a'
>>> arr.mean('a')
b  b0  b1  b2  b3
   6.0 7.0 8.0 9.0
>>> # along axis 'b'
>>> arr.mean('b')
a  a0  a1  a2  a3
   1.5 5.5 9.5 13.5
```

Select some rows only

```
>>> arr.mean(['a0', 'a1'])
b  b0  b1  b2  b3
   2.0 3.0 4.0 5.0
>>> # or equivalently
>>> # arr.mean('a0,a1')
```

Split an axis in several parts

```
>>> arr.mean(['a0', 'a1'], ['a2', 'a3'])
a\b  b0  b1  b2  b3
a0,a1 2.0  3.0  4.0  5.0
a2,a3 10.0 11.0 12.0 13.0
>>> # or equivalently
>>> # arr.mean('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.mean((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b  b0  b1  b2  b3
a01  2.0  3.0  4.0  5.0
a23 10.0 11.0 12.0 13.0
>>> # or equivalently
>>> # arr.mean('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.mean_by

`Array.mean_by(*axes_and_groups, dtype=None, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the arithmetic mean.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The mean is performed along all axes except the given one(s). For groups, mean is performed along groups and non associated axes. The default (no axis or group) is to perform the mean over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the mean over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

Array.mean, Array.median, Array.median_by
Array.var, Array.var_by, Array.std, Array.std_by
Array.percentile, Array.percentile_by

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.mean()
7.5
>>> # along axis 'a'
>>> arr.mean_by('a')
a  a0  a1  a2  a3
   1.5 5.5 9.5 13.5
>>> # along axis 'b'
>>> arr.mean_by('b')
b  b0  b1  b2  b3
   6.0 7.0 8.0 9.0
```

Select some rows only

```
>>> arr.mean_by(['a0', 'a1'])
3.5
>>> # or equivalently
>>> # arr.mean_by('a0,a1')
```

Split an axis in several parts

```
>>> arr.mean_by(['a0', 'a1'], ['a2', 'a3'])
a  a0,a1  a2,a3
   3.5   11.5
>>> # or equivalently
>>> # arr.mean_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.mean_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01  a23
   3.5  11.5
>>> # or equivalently
>>> # arr.mean_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.median

`Array.median(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the arithmetic median.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the median is performed. The default (no axis or group) is to perform the median over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the median over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[`Array.median_by`](#), [`Array.mean`](#), [`Array.mean_by`](#)

[`Array.var`](#), [`Array.var_by`](#), [`Array.std`](#), [`Array.std_by`](#)

[`Array.percentile`](#), [`Array.percentile_by`](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr[:, :] = [[10, 7, 5, 9],
...              [5, 8, 3, 7],
...              [6, 2, 0, 9],
...              [9, 10, 5, 6]]
>>> arr
a\b  b0  b1  b2  b3
a0   10   7   5   9
a1    5   8   3   7
a2    6   2   0   9
a3    9  10   5   6
>>> arr.median()
6.5
>>> # along axis 'a'
>>> arr.median('a')
b  b0  b1  b2  b3
   7.5 7.5 4.0 8.0
>>> # along axis 'b'
>>> arr.median('b')
a  a0  a1  a2  a3
   8.0 6.0 4.0 7.5
```

Select some rows only

```
>>> arr.median(['a0', 'a1'])
b  b0  b1  b2  b3
   7.5 7.5 4.0 8.0
>>> # or equivalently
>>> # arr.median('a0,a1')
```

Split an axis in several parts

```
>>> arr.median(['a0', 'a1'], ['a2', 'a3'])
a\b  b0  b1  b2  b3
a0,a1 7.5 7.5 4.0 8.0
a2,a3 7.5 6.0 2.5 7.5
>>> # or equivalently
>>> # arr.median('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.median((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b  b0  b1  b2  b3
a01 7.5 7.5 4.0 8.0
a23 7.5 6.0 2.5 7.5
>>> # or equivalently
>>> # arr.median('a0,a1>>a01;a2,a3>>a23')
```


larray.Array.median_by

`Array.median_by(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the arithmetic median.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The median is performed along all axes except the given one(s). For groups, median is performed along groups and non associated axes. The default (no axis or group) is to perform the median over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the median over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[`Array.median`](#), [`Array.mean`](#), [`Array.mean_by`](#)

[`Array.var`](#), [`Array.var_by`](#), [`Array.std`](#), [`Array.std_by`](#)

[`Array.percentile`](#), [`Array.percentile_by`](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr[:, :] = [[10, 7, 5, 9],
...              [5, 8, 3, 7],
...              [6, 2, 0, 9],
...              [9, 10, 5, 6]]
>>> arr
a\b  b0  b1  b2  b3
a0   10   7   5   9
a1    5   8   3   7
a2    6   2   0   9
a3    9  10   5   6
>>> arr.median_by()
6.5
>>> # along axis 'a'
>>> arr.median_by('a')
a  a0  a1  a2  a3
   8.0 6.0 4.0 7.5
>>> # along axis 'b'
>>> arr.median_by('b')
b  b0  b1  b2  b3
   7.5 7.5 4.0 8.0
```

Select some rows only

```
>>> arr.median_by(['a0', 'a1'])
7.0
>>> # or equivalently
>>> # arr.median_by('a0,a1')
```

Split an axis in several parts

```
>>> arr.median_by((['a0', 'a1'], ['a2', 'a3']))
a  a0,a1  a2,a3
   7.0   5.75
>>> # or equivalently
>>> # arr.median_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.median_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01  a23
   7.0  5.75
>>> # or equivalently
>>> # arr.median_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.var

`Array.var(*axes_and_groups, dtype=None, ddof=1, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the unbiased variance.

Normalized by N-1 by default. This can be changed using the `ddof` argument.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the variance is performed. The default (no axis or group) is to perform the variance over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the variance over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

ddof [int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$, where N represents the number of elements. Defaults to 1.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. `keepaxes='summation'`). Defaults to False.

Returns

Array or scalar

See also:

`Array.var_by`, `Array.std`, `Array.std_by`
`Array.mean`, `Array.mean_by`, `Array.median`, `Array.median_by`
`Array.percentile`, `Array.percentile_by`

Examples

```
>>> arr = ndtest((2, 8), dtype=float)
>>> arr[:, :] = [[0, 3, 5, 6, 4, 2, 1, 3],
...              [7, 3, 2, 5, 8, 5, 6, 4]]
>>> arr
a\b   b0   b1   b2   b3   b4   b5   b6   b7
a0   0.0  3.0  5.0  6.0  4.0  2.0  1.0  3.0
a1   7.0  3.0  2.0  5.0  8.0  5.0  6.0  4.0
>>> arr.var()
4.7999999999999998
>>> # along axis 'b'
>>> arr.var('b')
a   a0   a1
   4.0  4.0
```

Select some columns only

```
>>> arr.var(['b0', 'b1', 'b3'])
a   a0   a1
   9.0  4.0
>>> # or equivalently
>>> # arr.var('b0,b1,b3')
```

Split an axis in several parts

```
>>> arr.var(['b0', 'b1', 'b3'], 'b5:')
a\b   b0,b1,b3   b5:
a0         9.0   1.0
a1         4.0   1.0
>>> # or equivalently
>>> # arr.var('b0,b1,b3;b5:')
```

Same with renaming

```
>>> arr.var((X.b['b0', 'b1', 'b3'] >> 'b013', X.b['b5:'] >> 'b567'))
a\b   b013   b567
a0     9.0    1.0
a1     4.0    1.0
>>> # or equivalently
>>> # arr.var('b0,b1,b3>>b013;b5:>>b567')
```

larray.Array.var_by

`Array.var_by(*axes_and_groups, dtype=None, ddof=1, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the unbiased variance.

Normalized by N-1 by default. This can be changed using the `ddof` argument.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The variance is performed along all axes except the given one(s). For groups, variance is performed along groups and non associated axes. The default (no axis or group) is to perform the variance over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the variance over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

ddof [int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$, where N represents the number of elements. Defaults to 1.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. `keepaxes='summation'`). Defaults to False.

Returns

Array or scalar

See also:

Array.var, Array.std, Array.std_by

Array.mean, Array.mean_by, Array.median, Array.median_by

Array.percentile, Array.percentile_by

Examples

```
>>> arr = ndtest((2, 8), dtype=float)
>>> arr[:, :] = [[0, 3, 5, 6, 4, 2, 1, 3],
...              [7, 3, 2, 5, 8, 5, 6, 4]]
>>> arr
a\b   b0   b1   b2   b3   b4   b5   b6   b7
a0  0.0  3.0  5.0  6.0  4.0  2.0  1.0  3.0
a1  7.0  3.0  2.0  5.0  8.0  5.0  6.0  4.0
>>> arr.var_by()
4.7999999999999998
>>> # along axis 'a'
>>> arr.var_by('a')
a   a0   a1
   4.0  4.0
```

Select some columns only

```
>>> arr.var_by('a', ['b0', 'b1', 'b3'])
a   a0   a1
   9.0  4.0
>>> # or equivalently
>>> # arr.var_by('a', 'b0,b1,b3')
```

Split an axis in several parts

```
>>> arr.var_by('a', (['b0', 'b1', 'b3'], 'b5:'))
a\b   b0,b1,b3   b5:
a0         9.0  1.0
a1         4.0  1.0
>>> # or equivalently
>>> # arr.var_by('a', 'b0,b1,b3;b5:')
```

Same with renaming

```
>>> arr.var_by('a', (X.b['b0', 'b1', 'b3'] >> 'b013', X.b['b5:'] >> 'b567'))
a\b   b013   b567
a0     9.0    1.0
a1     4.0    1.0
>>> # or equivalently
>>> # arr.var_by('a', 'b0,b1,b3>>b013;b5:>>b567')
```

larray.Array.std

`Array.std(*axes_and_groups, dtype=None, ddof=1, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the sample standard deviation.

Normalized by N-1 by default. This can be changed using the `ddof` argument.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the standard deviation is performed. The default (no axis or group) is to perform the standard deviation over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the standard deviation over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

ddof [int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$, where N represents the number of elements. Defaults to 1.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. `keepaxes='summation'`). Defaults to False.

Returns

Array or scalar

See also:

`Array.std_by`, `Array.var`, `Array.var_by`

`Array.mean`, `Array.mean_by`, `Array.median`, `Array.median_by`

`Array.percentile`, `Array.percentile_by`

Examples

```
>>> arr = ndtest((2, 8), dtype=float)
>>> arr[:, :] = [[0, 3, 5, 6, 4, 2, 1, 3],
...              [7, 3, 2, 5, 8, 5, 6, 4]]
>>> arr
a\b   b0   b1   b2   b3   b4   b5   b6   b7
a0    0.0  3.0  5.0  6.0  4.0  2.0  1.0  3.0
a1    7.0  3.0  2.0  5.0  8.0  5.0  6.0  4.0
>>> arr.std()
2.1908902300206643
>>> # along axis 'b'
>>> arr.std('b')
a   a0   a1
   2.0  2.0
```

Select some columns only

```
>>> arr.std(['b0', 'b1', 'b3'])
a   a0   a1
   3.0  2.0
>>> # or equivalently
>>> # arr.std('b0,b1,b3')
```

Split an axis in several parts

```
>>> arr.std(['b0', 'b1', 'b3'], 'b5:')
a\b   b0,b1,b3   b5:
a0         3.0   1.0
a1         2.0   1.0
>>> # or equivalently
>>> # arr.std('b0,b1,b3;b5:')
```

Same with renaming

```
>>> arr.std((X.b['b0', 'b1', 'b3'] >> 'b013', X.b['b5:'] >> 'b567'))
a\b   b013   b567
a0     3.0    1.0
a1     2.0    1.0
>>> # or equivalently
>>> # arr.std('b0,b1,b3>>b013;b5:>>b567')
```


larray.Array.std_by

`Array.std_by(*axes_and_groups, dtype=None, ddof=1, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Computes the sample standard deviation.

Normalized by N-1 by default. This can be changed using the ddof argument.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The standard deviation is performed along all axes except the given one(s). For groups, standard deviation is performed along groups and non associated axes. The default (no axis or group) is to perform the standard deviation over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the standard deviation over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

dtype [dtype, optional] The data type of the returned array. Defaults to None (the dtype of the input array).

ddof [int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is $N - \text{ddof}$, where N represents the number of elements. Defaults to 1.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

Array.std_by, Array.var, Array.var_by

Array.mean, Array.mean_by, Array.median, Array.median_by

Array.percentile, Array.percentile_by

Examples

```
>>> arr = ndtest((2, 8), dtype=float)
>>> arr[:, :] = [[0, 3, 5, 6, 4, 2, 1, 3],
...              [7, 3, 2, 5, 8, 5, 6, 4]]
>>> arr
a\b   b0   b1   b2   b3   b4   b5   b6   b7
a0    0.0  3.0  5.0  6.0  4.0  2.0  1.0  3.0
a1    7.0  3.0  2.0  5.0  8.0  5.0  6.0  4.0
>>> arr.std_by()
2.1908902300206643
>>> # along axis 'a'
>>> arr.std_by('a')
a   a0   a1
   2.0  2.0
```

Select some columns only

```
>>> arr.std_by('a', ['b0', 'b1', 'b3'])
a   a0   a1
   3.0  2.0
>>> # or equivalently
>>> # arr.std_by('a', 'b0,b1,b3')
```

Split an axis in several parts

```
>>> arr.std_by('a', (['b0', 'b1', 'b3'], 'b5:'))
a\b   b0,b1,b3  b5:
a0         3.0  1.0
a1         2.0  1.0
>>> # or equivalently
>>> # arr.std_by('a', 'b0,b1,b3;b5:')
```

Same with renaming

```
>>> arr.std_by('a', (X.b['b0', 'b1', 'b3'] >> 'b013', X.b['b5:'] >> 'b567'))
a\b   b013  b567
a0     3.0   1.0
a1     2.0   1.0
>>> # or equivalently
>>> # arr.std_by('a', 'b0,b1,b3>>b013;b5:>>b567')
```

larray.Array.percentile

`Array.percentile(q, *axes_and_groups, out=None, interpolation='linear', skipna=None, keep-axes=False, **explicit_axes)`

Computes the qth percentile of the data along the specified axis.

Parameters

q [int in range of [0,100] (or sequence of floats)] Percentile to compute, which must be between 0 and 100 inclusive.

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the qth percentile is performed. The default (no axis or group) is to perform the qth percentile over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the qth percentile over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

interpolation [{ 'linear', 'lower', 'higher', 'midpoint', 'nearest' }, optional] Interpolation method to use when the desired quantile lies between two data points $i < j$:

- linear: $i + (j - i) * \text{fraction}$, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j*, whichever is nearest.
- midpoint: $(i + j) / 2$.

Defaults to 'linear'.

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

Array.percentile_by, Array.mean, Array.mean_by

Array.median, Array.median_by, Array.var, Array.var_by

Array.std, Array.std_by

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.percentile(25)
3.75
>>> # along axis 'a'
>>> arr.percentile(25, 'a')
b  b0  b1  b2  b3
   3.0 4.0 5.0 6.0
>>> # along axis 'b'
>>> arr.percentile(25, 'b')
a  a0  a1  a2  a3
   0.75 4.75 8.75 12.75
>>> # several percentile values
>>> arr.percentile([25, 50, 75], 'b')
percentile\a  a0  a1  a2  a3
            25  0.75 4.75 8.75 12.75
            50  1.5  5.5  9.5 13.5
            75  2.25 6.25 10.25 14.25
```

Select some rows only

```
>>> arr.percentile(25, ['a0', 'a1'])
b  b0  b1  b2  b3
   1.0 2.0 3.0 4.0
>>> # or equivalently
>>> # arr.percentile(25, 'a0,a1')
```

Split an axis in several parts

```
>>> arr.percentile(25, (['a0', 'a1'], ['a2', 'a3']))
a\b  b0  b1  b2  b3
```

(continues on next page)

(continued from previous page)

```

a0,a1  1.0   2.0   3.0   4.0
a2,a3  9.0  10.0  11.0  12.0
>>> # or equivalently
>>> # arr.percentile(25, 'a0,a1;a2,a3')

```

Same with renaming

```

>>> arr.percentile(25, (X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b    b0     b1     b2     b3
a01    1.0    2.0    3.0    4.0
a23    9.0   10.0   11.0   12.0
>>> # or equivalently
>>> # arr.percentile(25, 'a0,a1>>a01;a2,a3>>a23')

```

larray.Array.percentile_by

`Array.percentile_by(q, *axes_and_groups, out=None, interpolation='linear', skipna=None, keep-axes=False, **explicit_axes)`

Computes the qth percentile of the data for the specified axis.

Parameters

q [int in range of [0,100] (or sequence of floats)] Percentile to compute, which must be between 0 and 100 inclusive.

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The qth percentile is performed along all axes except the given one(s). For groups, qth percentile is performed along groups and non associated axes. The default (no axis or group) is to perform the qth percentile over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the qth percentile over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

interpolation [{ 'linear', 'lower', 'higher', 'midpoint', 'nearest' }, optional] Interpolation method to use when the desired quantile lies between two data points $i < j$:

- linear: $i + (j - i) * \text{fraction}$, where `fraction` is the fractional part of the index surrounded by i and j .
- lower: i .
- higher: j .
- nearest: i or j , whichever is nearest.
- midpoint: $(i + j) / 2$.

Defaults to 'linear'.

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. `keepaxes='summation'`). Defaults to False.

Returns

Array or scalar

See also:

Array.percentile, Array.mean, Array.mean_by

Array.median, Array.median_by, Array.var, Array.var_by

Array.std, Array.std_by

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.percentile_by(25)
3.75
>>> # along axis 'a'
>>> arr.percentile_by(25, 'a')
a   a0   a1   a2   a3
   0.75  4.75  8.75 12.75
>>> # along axis 'b'
>>> arr.percentile_by(25, 'b')
b   b0  b1  b2  b3
   3.0  4.0  5.0  6.0
>>> # several percentile values
```

(continues on next page)

(continued from previous page)

```
>>> arr.percentile_by([25, 50, 75], 'b')
percentile\b   b0    b1    b2    b3
              25  3.0   4.0   5.0   6.0
              50  6.0   7.0   8.0   9.0
              75  9.0  10.0  11.0  12.0
```

Select some rows only

```
>>> arr.percentile_by(25, ['a0', 'a1'])
1.75
>>> # or equivalently
>>> # arr.percentile_by('a0,a1')
```

Split an axis in several parts

```
>>> arr.percentile_by(25, (['a0', 'a1'], ['a2', 'a3']))
a  a0,a1  a2,a3
   1.75   9.75
>>> # or equivalently
>>> # arr.percentile_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.percentile_by(25, (X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01  a23
   1.75  9.75
>>> # or equivalently
>>> # arr.percentile_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.ptp

`Array.ptp(*axes_and_groups, out=None, **explicit_axes)`

Returns the range of values (maximum - minimum).

The name of the function comes from the acronym for ‘peak to peak’.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] Axis(es) or group(s) along which the ptp is performed. The default (no axis or group) is to perform the ptp over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string (‘axis_name’) or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the ptp over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- ([‘a1’, ‘a3’, ‘a5’], ‘b1, b3, b5’) : labels separated by commas in a list or a string

- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

Returns

Array or scalar

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.ptp()
15
>>> # along axis 'a'
>>> arr.ptp('a')
b  b0  b1  b2  b3
   12  12  12  12
>>> # along axis 'b'
>>> arr.ptp('b')
a  a0  a1  a2  a3
   3   3   3   3
```

Select some rows only

```
>>> arr.ptp(['a0', 'a1'])
b  b0  b1  b2  b3
   4   4   4   4
>>> # or equivalently
>>> # arr.ptp('a0,a1')
```

Split an axis in several parts

```
>>> arr.ptp(['a0', 'a1'], ['a2', 'a3'])
a\b  b0  b1  b2  b3
a0,a1  4   4   4   4
a2,a3  4   4   4   4
>>> # or equivalently
>>> # arr.ptp('a0,a1;a2,a3')
```

Same with renaming


```
>>> arr.ptp((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b  b0  b1  b2  b3
a01   4   4   4   4
a23   4   4   4   4
>>> # or equivalently
>>> # arr.ptp('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.with_total

Array.**with_total** (*args, op=sum, label='total', **kwargs)

Add aggregated values (sum by default) along each axis. A user defined label can be given to specified the computed values.

Parameters

***args** [int or str or Axis or Group or any combination of those, optional] Axes or groups along which to compute the aggregates. Passed groups should be named. Defaults to aggregate over the whole array.

op [aggregate function, optional] Available aggregate functions are: *sum*, *prod*, *min*, *max*, *mean*, *ptp*, *var*, *std*, *median* and *percentile*. Defaults to *sum*.

label [scalar value, optional] Label to use for the total. Applies only to aggregated axes, not groups. Defaults to “total”.

****kwargs** [int or str or Group or any combination of those, optional] Axes or groups along which to compute the aggregates.

Returns

Array

Examples

```
>>> arr = ndtest("gender=M,F;time=2013..2016")
>>> arr
gender\time  2013  2014  2015  2016
           M     0     1     2     3
           F     4     5     6     7
>>> arr.with_total()
gender\time  2013  2014  2015  2016  total
           M     0     1     2     3     6
           F     4     5     6     7    22
    total    4     6     8    10    28
```

Using another function and label

```
>>> arr.with_total(op=mean, label='mean')
gender\time  2013  2014  2015  2016  mean
           M   0.0   1.0   2.0   3.0   1.5
           F   4.0   5.0   6.0   7.0   5.5
    mean     2.0   3.0   4.0   5.0   3.5
```

Specifying an axis and a label

```
>>> arr.with_total('gender', label='U')
gender\time 2013 2014 2015 2016
           M    0    1    2    3
           F    4    5    6    7
           U    4    6    8   10
```

Using groups

```
>>> time_groups = (arr.time[:2014] >> 'before_2015',
...               arr.time[2015:] >> 'after_2015')
>>> arr.with_total(time_groups)
gender\time 2013 2014 2015 2016 before_2015 after_2015
           M    0    1    2    3           1           5
           F    4    5    6    7           9          13
>>> # or equivalently
>>> # arr.with_total('time[:2014] >> before_2015; time[2015:] >> after_2015')
```

larray.Array.percent

Array.**percent** (*self*, *axes)

Returns an array with values given as percent of the total of all values along given axes.

Parameters

*axes

Returns

Array array / array.sum(axes) * 100

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> a = Array([[4, 6], [2, 8]], [nat, sex])
>>> a
nat\sex  M  F
      BE  4  6
      FO  2  8
>>> a.percent()
nat\sex  M    F
      BE 20.0 30.0
      FO 10.0 40.0
>>> a.percent('sex')
nat\sex  M    F
      BE 40.0 60.0
      FO 20.0 80.0
```

larray.Array.ratio

Array.**ratio**(*self*, **axes*)

Returns an array with all values divided by the sum of values along given axes.

Parameters

***axes**

Returns

Array array / array.sum(axes)

Examples

```

>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> a = Array([[4, 6], [2, 8]], [nat, sex])
>>> a
nat\sex  M  F
      BE  4  6
      FO  2  8
>>> a.sum()
20
>>> a.ratio()
nat\sex    M    F
      BE  0.2  0.3
      FO  0.1  0.4
>>> a.ratio('sex')
nat\sex    M    F
      BE  0.4  0.6
      FO  0.2  0.8
>>> a.ratio('M')
nat\sex    M    F
      BE  1.0  1.5
      FO  1.0  4.0

```

larray.Array.rationot0

Array.**rationot0**(*self*, **axes*)

Returns an Array with values array / array.sum(axes) where the sum is not 0, 0 otherwise.

Parameters

***axes**

Returns

Array array / array.sum(axes)

Examples

```
>>> a = Axis('a=a0,a1')
>>> b = Axis('b=b0,b1,b2')
>>> arr = Array([[6, 0, 2],
...              [4, 0, 8]], [a, b])
>>> arr
a\b  b0  b1  b2
a0   6   0   2
a1   4   0   8
>>> arr.sum()
20
>>> arr.rationot0()
a\b  b0  b1  b2
a0  0.3  0.0  0.1
a1  0.2  0.0  0.4
>>> arr.rationot0('a')
a\b  b0  b1  b2
a0  0.6  0.0  0.2
a1  0.4  0.0  0.8
```

for reference, the normal ratio method would return:

```
>>> arr.ratio('a')
a\b  b0  b1  b2
a0  0.6  nan  0.2
a1  0.4  nan  0.8
```

larray.Array.growth_rate

`Array.growth_rate` (*self*, *axis=-1*, *d=1*, *label='upper'*)

Calculates the growth along a given axis.

Roughly equivalent to `a.diff(axis, d, label) / a[axis.i[:-d]]`

Parameters

axis [int, str, Group or Axis, optional] Axis or group along which the difference is taken. Defaults to the last axis.

d [int, optional] Periods to shift for forming difference. Defaults to 1.

label [{ 'lower', 'upper' }, optional] The new labels in *axis* will have the labels of either the array being subtracted ('lower') or the array it is subtracted from ('upper'). Defaults to 'upper'.

Returns

Array

Examples

```
>>> data = [[2, 4, 5, 4, 6], [4, 6, 3, 6, 9]]
>>> a = Array(data, "sex=M,F; year=2016..2020")
>>> a
sex\year  2016  2017  2018  2019  2020
      M     2     4     5     4     6
      F     4     6     3     6     9
>>> a.growth_rate()
sex\year  2017  2018  2019  2020
      M   1.0  0.25 -0.2   0.5
      F   0.5 -0.5   1.0   0.5
>>> a.growth_rate(label='lower')
sex\year  2016  2017  2018  2019
      M   1.0  0.25 -0.2   0.5
      F   0.5 -0.5   1.0   0.5
>>> a.growth_rate(d=2)
sex\year  2018  2019  2020
      M   1.5   0.0   0.2
      F -0.25   0.0   2.0
>>> a.growth_rate('sex')
sex\year  2016  2017  2018  2019  2020
      F   1.0   0.5 -0.4   0.5   0.5
>>> a.growth_rate(a.year[2017:])
sex\year  2018  2019  2020
      M  0.25 -0.2   0.5
      F -0.5   1.0   0.5
```

larray.Array.describe

`Array.describe` (*self*, *args, **kwargs)

Descriptive summary statistics, excluding NaN values.

By default, it includes the number of non-NaN values, the mean, standard deviation, minimum, maximum and the 25, 50 and 75 percentiles.

Parameters

***args** [int or str or Axis or Group or any combination of those, optional] Axes or groups along which to compute the aggregates. Defaults to aggregate over the whole array.

percentiles [array-like, optional.] List of integer percentiles to include. Defaults to [25, 50, 75].

Returns

Array

See also:

[`Array.describe_by`](#)

Examples

```
>>> arr = Array([0, 6, 2, 5, 4, 3, 1, 3], 'year=2013..2020')
>>> arr
year  2013  2014  2015  2016  2017  2018  2019  2020
      0     6     2     5     4     3     1     3
>>> arr.describe()
statistic count mean std min 25% 50% 75% max
          8.0  3.0  2.0  0.0  1.75  3.0  4.25  6.0
>>> arr.describe(percentiles=[50, 90])
statistic count mean std min 50% 90% max
          8.0  3.0  2.0  0.0  3.0  5.3  6.0
```

larray.Array.describe_by

`Array.describe_by(self, *args, **kwargs)`

Descriptive summary statistics, excluding NaN values, along axes or for groups.

By default, it includes the number of non-NaN values, the mean, standard deviation, minimum, maximum and the 25, 50 and 75 percentiles.

Parameters

***args** [int or str or Axis or Group or any combination of those, optional] Axes or groups to include in the result after aggregating. Defaults to aggregate over the whole array.

percentiles [array-like, optional.] list of integer percentiles to include. Defaults to [25, 50, 75].

Returns

Array

See also:

[`Array.describe`](#)

Examples

```
>>> data = [[0, 6, 3, 5, 4, 2, 1, 3], [7, 5, 3, 2, 8, 5, 6, 4]]
>>> arr = Array(data, 'gender=Male,Female;year=2013..2020').astype(float)
>>> arr
gender\year  2013  2014  2015  2016  2017  2018  2019  2020
      Male   0.0   6.0   3.0   5.0   4.0   2.0   1.0   3.0
      Female  7.0   5.0   3.0   2.0   8.0   5.0   6.0   4.0
>>> arr.describe_by('gender')
gender\statistic count mean std min 25% 50% 75% max
      Male       8.0  3.0  2.0  0.0  1.75  3.0  4.25  6.0
      Female     8.0  5.0  2.0  2.0  3.75  5.0  6.25  8.0
>>> arr.describe_by('gender', (X.year[:2015], X.year[2018:]))
gender year\statistic count mean std min 25% 50% 75% max
      Male      :2015   3.0  3.0  3.0  0.0  1.5  3.0  4.5  6.0
      Male      2018:   3.0  2.0  1.0  1.0  1.5  2.0  2.5  3.0
      Female      :2015   3.0  5.0  2.0  3.0  4.0  5.0  6.0  7.0
      Female      2018:   3.0  5.0  1.0  4.0  4.5  5.0  5.5  6.0
>>> arr.describe_by('gender', percentiles=[50, 90])
gender\statistic count mean std min 50% 90% max
```

(continues on next page)

(continued from previous page)

Male	8.0	3.0	2.0	0.0	3.0	5.3	6.0
Female	8.0	5.0	2.0	2.0	5.0	7.3	8.0

Sorting

<code>Array.sort_axes(self[, axes, ascending])</code>	Sorts axes of the array.
<code>Array.sort_values(self[, key, axis, ascending])</code>	Sorts values of the array.
<code>Array.labelsofsorted(self[, axis, ...])</code>	Returns the labels that would sort this array.
<code>Array.indicesofsorted(self[, axis, ...])</code>	Returns the indices that would sort this array.

`larray.Array.sort_axes`

`Array.sort_axes` (*self*, *axes=None*, *ascending=True*)

Sorts axes of the array.

Parameters

axes [axis reference (Axis, str, int) or list of them, optional] Axes to sort. Defaults to all axes.

ascending [bool, optional] Sort axes in ascending order. Defaults to True.

Returns

Array Array with sorted axes.

Examples

```
>>> a = ndtest("nat=EU,FO,BE; sex=M,F")
>>> a
nat\sex  M  F
      EU  0  1
      FO  2  3
      BE  4  5
>>> a.sort_axes('sex')
nat\sex  F  M
      EU  1  0
      FO  3  2
      BE  5  4
>>> a.sort_axes()
nat\sex  F  M
      BE  5  4
      EU  1  0
      FO  3  2
>>> a.sort_axes(('sex', 'nat'))
nat\sex  F  M
      BE  5  4
      EU  1  0
      FO  3  2
>>> a.sort_axes(ascending=False)
nat\sex  M  F
      FO  2  3
      EU  0  1
      BE  4  5
```

larray.Array.sort_values

`Array.sort_values(self, key=None, axis=None, ascending=True)`

Sorts values of the array.

Parameters

key [scalar or tuple or Group] Key along which to sort. Must have exactly one dimension less than `ndim`. Cannot be used in combination with `axis` argument. If both `key` and `axis` are `None`, sort array with all axes combined. Defaults to `None`.

axis [int or str or Axis] Axis along which to sort. Cannot be used in combination with `key` argument. Defaults to `None`.

ascending [bool, optional] Sort values in ascending order. Defaults to `True`.

Returns

Array Array with sorted values.

Examples

sort the whole array (no key or axis given)

```

>>> arr_1D = Array([10, 2, 4], 'a=a0..a2')
>>> arr_1D
a  a0  a1  a2
   10   2   4
>>> arr_1D.sort_values()
a  a1  a2  a0
   2   4  10
>>> arr_2D = Array([[10, 2, 4], [3, 7, 1]], 'a=a0,a1; b=b0..b2')
>>> arr_2D
a\b  b0  b1  b2
a0   10   2   4
a1    3   7   1
>>> # if the array has more than one dimension, sort array with all axes combined
>>> arr_2D.sort_values()
a_b  a1_b2  a0_b1  a1_b0  a0_b2  a1_b1  a0_b0
     1       2       3       4       7      10

```

Sort along a given key

```

>>> # sort columns according to the values of the row associated with the label
    ↪ 'a1'
>>> arr_2D.sort_values('a1')
a\b  b2  b0  b1
a0    4  10   2
a1    1   3   7
>>> arr_2D.sort_values('a1', ascending=False)
a\b  b1  b0  b2
a0    2  10   4
a1    7   3   1
>>> arr_3D = Array([[[10, 2, 4], [3, 7, 1]], [[5, 1, 6], [2, 8, 9]]],
...                  'a=a0,a1; b=b0,b1; c=c0..c2')
>>> arr_3D
a  b\c  c0  c1  c2
a0   b0  10   2   4

```

(continues on next page)

(continued from previous page)

```

a0  b1  3  7  1
a1  b0  5  1  6
a1  b1  2  8  9
>>> # sort columns according to the values of the row associated with the labels
    ↪ 'a0' and 'b1'
>>> arr_3D.sort_values(('a0', 'b1'))
  a  b\c  c2  c0  c1
a0  b0   4  10   2
a0  b1   1   3   7
a1  b0   6   5   1
a1  b1   9   2   8

```

Sort along an axis

```

>>> arr_2D
a\b  b0  b1  b2
a0  10   2   4
a1   3   7   1
>>> # sort values along axis 'a'
>>> # equivalent to sorting the values of each column of the array
>>> arr_2D.sort_values(axis='a')
a*\b  b0  b1  b2
0     3   2   1
1    10   7   4
>>> # sort values along axis 'b'
>>> # equivalent to sorting the values of each row of the array
>>> arr_2D.sort_values(axis='b')
a\b*  0  1  2
a0    2  4 10
a1    1  3  7

```

larray.Array.labelsofsorted

`Array.labelsofsorted(self, axis=None, ascending=True, kind='quicksort')`

Returns the labels that would sort this array.

Performs an indirect sort along the given axis using the algorithm specified by the *kind* keyword. It returns an array of labels of the same shape as *a* that index data along the given axis in sorted order.

Parameters

axis [int or str or Axis, optional] Axis along which to sort. This can be omitted if array has only one axis.

ascending [bool, optional] Sort values in ascending order. Defaults to True.

kind [{‘quicksort’, ‘mergesort’, ‘heapsort’}, optional] Sorting algorithm. Defaults to ‘quicksort’.

Returns

Array

Examples

```
>>> arr = Array([[0, 1], [3, 2], [2, 5]], "nat=BE,FR,IT; sex=M,F")
>>> arr
nat\sex  M  F
      BE  0  1
      FR  3  2
      IT  2  5
>>> arr.labelsofsorted('sex')
nat\sex  0  1
      BE  M  F
      FR  F  M
      IT  M  F
>>> arr.labelsofsorted('sex', ascending=False)
nat\sex  0  1
      BE  F  M
      FR  M  F
      IT  F  M
```

larray.Array.indicesofsorted

`Array.indicesofsorted(self, axis=None, ascending=True, kind='quicksort')`

Returns the indices that would sort this array.

Performs an indirect sort along the given axis using the algorithm specified by the *kind* keyword. It returns an array of indices with the same axes as *a* that index data along the given axis in sorted order.

Parameters

axis [int or str or Axis, optional] Axis along which to sort. This can be omitted if array has only one axis.

ascending [bool, optional] Sort values in ascending order. Defaults to True.

kind [{‘quicksort’, ‘mergesort’, ‘heapsort’}, optional] Sorting algorithm. Defaults to ‘quicksort’.

Returns

Array

Examples

```
>>> arr = Array([[1, 5], [3, 2], [0, 4]], "nat=BE,FR,IT; sex=M,F")
>>> arr
nat\sex  M  F
      BE  1  5
      FR  3  2
      IT  0  4
>>> arr.indicesofsorted('nat')
nat\sex  M  F
      0  2  1
      1  0  2
      2  1  0
>>> arr.indicesofsorted('nat', ascending=False)
nat\sex  M  F
```

(continues on next page)

(continued from previous page)

```

0  1  0
1  0  2
2  2  1

```

Reshaping/Extending/Reordering

<code>Array.reshape(self, target_axes)</code>	Given a list of new axes, changes the shape of the array.
<code>Array.reshape_like(self, target)</code>	Same as reshape but with an array as input.
<code>Array.compact(self)</code>	Detects and removes “useless” axes (ie axes for which values are constant over the whole axis)
<code>Array.reindex(self[, axes_to_reindex, ...])</code>	Reorder and/or add new labels in axes.
<code>Array.transpose(self, *args)</code>	Reorder axes.
<code>Array.expand(self[, target_axes, out, readonly])</code>	Expands array to target_axes.
<code>Array.prepend(self, axis, value[, label])</code>	Adds an array before self along an axis.
<code>Array.append(self, axis, value[, label])</code>	Adds an array to self along an axis.
<code>Array.extend(self, axis, other)</code>	Adds an array to self along an axis.
<code>Array.insert(self, value[, before, after, ...])</code>	Inserts value in array along an axis.
<code>Array.broadcast_with(self, target)</code>	Returns an array that is (NumPy) broadcastable with target.
<code>Array.align(self, other[, join, fill_value, ...])</code>	Align two arrays on their axes with the specified join method.

larray.Array.reshape

`Array.reshape(self, target_axes)`

Given a list of new axes, changes the shape of the array. The size of the array (= number of elements) must be equal to the product of length of target axes.

Parameters

target_axes [iterable of Axis] New axes. The size of the array (= number of stored data) must be equal to the product of length of target axes.

Returns

Array New array with new axes but same data.

Examples

```

>>> arr = ndtest((2, 2, 2))
>>> arr
a  b\c  c0  c1
a0  b0   0   1
a0  b1   2   3
a1  b0   4   5
a1  b1   6   7
>>> new_arr = arr.reshape([Axis('a=a0,a1'),
... Axis(['b0c0', 'b0c1', 'b1c0', 'b1c1'], 'bc')])
>>> new_arr
a\bc  b0c0  b0c1  b1c0  b1c1

```

(continues on next page)

(continued from previous page)

a0	0	1	2	3
a1	4	5	6	7

`larray.Array.reshape_like`

`Array.reshape_like(self, target)`

Same as `reshape` but with an array as input. Total size (= number of stored data) of the two arrays must be equal.

See also:

[`reshape`](#) returns an Array with a new shape given a list of axes.

Examples

```
>>> arr = zeros((2, 2, 2), dtype=int)
>>> arr
{0}* {1}*\{2}* 0 1
  0      0 0 0
  0      1 0 0
  1      0 0 0
  1      1 0 0
>>> new_arr = arr.reshape_like(ndtest((2, 4)))
>>> new_arr
a\b  b0  b1  b2  b3
a0   0   0   0   0
a1   0   0   0   0
```

`larray.Array.compact`

`Array.compact(self)`

Detects and removes “useless” axes (ie axes for which values are constant over the whole axis)

Returns

Array or scalar Array with constant axes removed.

Examples

```
>>> a = Array([[1, 2],
...           [1, 2]], [Axis('sex=M,F'), Axis('nat=BE,FO')])
>>> a
sex\nat  BE  FO
      M   1   2
      F   1   2
>>> a.compact()
nat  BE  FO
    1   2
```

larray.Array.reindex

`Array.reindex` (*self*, *axes_to_reindex*=None, *new_axis*=None, *fill_value*=nan, *inplace*=False, ***kwargs*)
 Reorder and/or add new labels in axes.

Place NaN or given *fill_value* in locations having no value previously.

Parameters

axes_to_reindex [axis ref or dict {axis ref: axis} or list of (axis ref, axis) or sequence of Axis]
 Axis(es) to reindex. If a single axis reference is given, the *new_axis* argument must be provided. If string, Group or Axis object, the corresponding axis is reindexed if found among existing, otherwise a new axis is added. If a list of Axis or an AxisCollection is given, existing axes are reindexed while missing ones are added.

new_axis [int, str, list/tuple/array of str, Group or Axis, optional] List of new labels or new axis if *axes_to_reindex* contains a single axis reference.

fill_value [scalar or Array, optional] Value used to fill cells corresponding to label combinations which were not present before reindexing. Defaults to NaN.

inplace [bool, optional] Whether or not to modify the original object or return a new array and leave the original intact. Defaults to False.

****kwargs** [Axis] New axis for each axis to reindex given as a keyword argument.

Returns

Array Array with reindexed axes.

Notes

When introducing NaNs into an array containing integers via reindex, all data will be promoted to float in order to store the NaNs.

Examples

```
>>> arr = ndtest((2, 2))
>>> arr
a\b  b0  b1
a0    0   1
a1    2   3
>>> arr2 = ndtest('a=a1,a2;c=c0;b=b2..b0')
>>> arr2
a  c\b  b2  b1  b0
a1  c0   0   1   2
a2  c0   3   4   5
```

Reindex an axis by passing labels (list or string)

```
>>> arr.reindex('b', ['b1', 'b2', 'b0'])
a\b  b1  b2  b0
a0  1.0 nan  0.0
a1  3.0 nan  2.0
>>> arr.reindex('b', 'b0..b2', fill_value=-1)
a\b  b0  b1  b2
a0   0   1  -1
```

(continues on next page)

(continued from previous page)

```

a1  2  3 -1
>>> arr.reindex(b='b=b0..b2', fill_value=-1)
a\b  b0  b1  b2
a0   0  1 -1
a1   2  3 -1

```

Reindex using an axis from another array

```

>>> arr.reindex('b', arr2.b, fill_value=-1)
a\b  b2  b1  b0
a0  -1  1  0
a1  -1  3  2

```

Reindex using a subset of an axis

```

>>> arr.reindex('b', arr2.b['b1':], fill_value=-1)
a\b  b1  b0
a0   1  0
a1   3  2

```

Reindex by passing an axis or a group

```

>>> arr.reindex('b=b2..b0', fill_value=-1)
a\b  b2  b1  b0
a0  -1  1  0
a1  -1  3  2
>>> arr.reindex(arr2.b, fill_value=-1)
a\b  b2  b1  b0
a0  -1  1  0
a1  -1  3  2
>>> arr.reindex(arr2.b['b1':], fill_value=-1)
a\b  b1  b0
a0   1  0
a1   3  2

```

Reindex several axes

```

>>> arr.reindex({'a': arr2.a, 'b': arr2.b}, fill_value=-1)
a\b  b2  b1  b0
a1  -1  3  2
a2  -1 -1 -1
>>> arr.reindex({'a': arr2.a, 'b': arr2.b['b1':]}, fill_value=-1)
a\b  b1  b0
a1   3  2
a2  -1 -1
>>> arr.reindex(a=arr2.a, b=arr2.b, fill_value=-1)
a\b  b2  b1  b0
a1  -1  3  2
a2  -1 -1 -1

```

Reindex by passing a collection of axes

```

>>> arr.reindex(arr2.axes, fill_value=-1)
a  b\c  c0
a1  b2 -1
a1  b1  3

```

(continues on next page)

(continued from previous page)

```

a1  b0  2
a2  b2  -1
a2  b1  -1
a2  b0  -1
>>> arr2.reindex(arr.axes, fill_value=-1)
a  c\b  b0  b1
a0  c0  -1  -1
a1  c0  2   1

```

larray.Array.transpose

`Array.transpose(self, *args)`

Reorder axes.

By default, reverse axes, otherwise permute the axes according to the list given as argument.

Parameters

***args** Accepts either a tuple of axes specs or axes specs as **args*. Omitted axes keep their order. Use ... to avoid specifying intermediate axes.

Returns

Array Array with reordered axes.

Examples

```

>>> arr = ndtest((2, 2, 2))
>>> arr
a  b\c  c0  c1
a0  b0  0   1
a0  b1  2   3
a1  b0  4   5
a1  b1  6   7
>>> arr.transpose('b', 'c', 'a')
b  c\a  a0  a1
b0  c0  0   4
b0  c1  1   5
b1  c0  2   6
b1  c1  3   7
>>> arr.transpose('b')
b  a\c  c0  c1
b0  a0  0   1
b0  a1  4   5
b1  a0  2   3
b1  a1  6   7
>>> arr.transpose(..., 'a')
b  c\a  a0  a1
b0  c0  0   4
b0  c1  1   5
b1  c0  2   6
b1  c1  3   7
>>> arr.transpose('c', ..., 'a')
c  b\a  a0  a1
c0  b0  0   4

```

(continues on next page)

(continued from previous page)

c0	b1	2	6
c1	b0	1	5
c1	b1	3	7

larray.Array.expand

Array.**expand**(*self*, *target_axes=None*, *out=None*, *readonly=False*)

Expands array to *target_axes*.

Target axes will be added to array if not present. In most cases this function is not needed because Array can do operations with arrays having different (compatible) axes.

Parameters

target_axes [string, list of Axis or AxisCollection, optional] Self can contain axes not present in *target_axes*. The result axes will be: [self.axes not in *target_axes*] + *target_axes*

out [Array, optional] Output array, must have more axes than array. Defaults to a new array. `arr.expand(out=out)` is equivalent to `out[:] = arr`

readonly [bool, optional] Whether returning a readonly view is acceptable or not (this is much faster)

Returns

Array Original array if possible (and out is None).

Examples

```
>>> a = Axis('a=a1,a2')
>>> b = Axis('b=b1,b2')
>>> arr = ndtest([a, b])
>>> arr
a\b  b1  b2
a1   0   1
a2   2   3
```

Adding one or several axes will append the new axes at the end

```
>>> c = Axis('c=c1,c2')
>>> arr.expand(c)
a  b\c  c1  c2
a1  b1   0   0
a1  b2   1   1
a2  b1   2   2
a2  b2   3   3
```

If you want new axes to be inserted in a particular order, you have to give that order

```
>>> arr.expand([a, c, b])
a  c\b  b1  b2
a1  c1   0   1
a1  c2   0   1
a2  c1   2   3
a2  c2   2   3
```


But it is enough to list only the added axes and the axes after them:

```
>>> arr.expand([c, b])
a  c\b  b1  b2
a1  c1   0   1
a1  c2   0   1
a2  c1   2   3
a2  c2   2   3
```

larray.Array.prepend

Array.**prepend** (*self*, *axis*, *value*, *label=None*)

Adds an array before self along an axis.

The two arrays must have compatible axes.

Parameters

axis [axis reference] Axis along which to prepend input array (*value*)

value [scalar or Array] Scalar or array with compatible axes.

label [str, optional] Label for the new item in axis

Returns

Array Array expanded with ‘value’ at the start of ‘axis’.

Examples

```
>>> a = ones('nat=BE,FO;sex=M,F')
>>> a
nat\sex    M    F
   BE  1.0  1.0
   FO  1.0  1.0
>>> a.prepend('sex', a.sum('sex'), 'M+F')
nat\sex  M+F    M    F
   BE  2.0  1.0  1.0
   FO  2.0  1.0  1.0
>>> a.prepend('nat', 2, 'Other')
nat\sex    M    F
   Other  2.0  2.0
   BE    1.0  1.0
   FO    1.0  1.0
>>> b = zeros('type=type1,type2')
>>> b
type  type1  type2
   0.0    0.0
>>> a.prepend('sex', b, 'Other')
nat  sex\type  type1  type2
   BE   Other    0.0    0.0
   BE     M     1.0    1.0
   BE     F     1.0    1.0
   FO   Other    0.0    0.0
   FO     M     1.0    1.0
   FO     F     1.0    1.0
```

larray.Array.append

`Array.append(self, axis, value, label=None)`

Adds an array to self along an axis.

The two arrays must have compatible axes.

Parameters

axis [axis reference] Axis along which to append *value*.

value [scalar or Array] Scalar or array with compatible axes.

label [scalar, optional] Label for the new item in axis

Returns

Array Array expanded with *value* along *axis*.

Examples

```
>>> a = ones('nat=BE,FO;sex=M,F')
>>> a
nat\sex    M    F
   BE  1.0  1.0
   FO  1.0  1.0
>>> a.append('sex', a.sum('sex'), 'M+F')
nat\sex    M    F  M+F
   BE  1.0  1.0  2.0
   FO  1.0  1.0  2.0
>>> a.append('nat', 2, 'Other')
nat\sex    M    F
   BE  1.0  1.0
   FO  1.0  1.0
   Other 2.0  2.0
>>> b = zeros('type=type1,type2')
>>> b
type  type1  type2
      0.0    0.0
>>> a.append('nat', b, 'Other')
nat  sex\type  type1  type2
   BE          M    1.0    1.0
   BE          F    1.0    1.0
   FO          M    1.0    1.0
   FO          F    1.0    1.0
Other          M    0.0    0.0
Other          F    0.0    0.0
```

larray.Array.extend

`Array.extend(self, axis, other)`

Adds an array to self along an axis.

The two arrays must have compatible axes.

Parameters

axis [axis] Axis along which to extend with input array (*other*)

other [Array] Array with compatible axes

Returns

Array Array expanded with ‘other’ along ‘axis’.

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> sex2 = Axis('sex=U')
>>> xtype = Axis('type=type1,type2')
>>> arr1 = ones([sex, xtype])
>>> arr1
sex\type  type1  type2
      M      1.0    1.0
      F      1.0    1.0
>>> arr2 = zeros([sex2, xtype])
>>> arr2
sex\type  type1  type2
      U      0.0    0.0
>>> arr1.extend('sex', arr2)
sex\type  type1  type2
      M      1.0    1.0
      F      1.0    1.0
      U      0.0    0.0
>>> arr3 = zeros([sex2, nat])
>>> arr3
sex\nat    BE    FO
      U  0.0  0.0
>>> arr1.extend('sex', arr3)
sex  type\nat    BE    FO
  M   type1  1.0  1.0
  M   type2  1.0  1.0
  F   type1  1.0  1.0
  F   type2  1.0  1.0
  U   type1  0.0  0.0
  U   type2  0.0  0.0
```

larray.Array.insert

`Array.insert` (*self*, *value*, *before=None*, *after=None*, *pos=None*, *axis=None*, *label=None*)

Inserts value in array along an axis.

Parameters

value [scalar or Array] Value to insert. If an Array, it must have compatible axes. If value already has the axis along which it is inserted, *label* should not be used.

before [scalar or Group] Label or group before which to insert *value*.

after [scalar or Group] Label or group after which to insert *value*.

label [str, optional] Label for the new item in axis.

Returns

Array Array with *value* inserted along *axis*. The dtype of the returned array will be the “closest” type which can hold both the array values and the inserted values without loss of information. For example, when mixing numeric and string types, the dtype will be object.

Examples

```
>>> arr1 = ndtest((2, 3))
>>> arr1
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
>>> arr1.insert(42, before='b1', label='b0.5')
a\b  b0  b0.5  b1  b2
a0    0    42   1   2
a1    3    42   4   5
```

The inserted value can be an array:

```
>>> arr2 = ndtest(2)
>>> arr2
a  a0  a1
   0   1
>>> arr1.insert(arr2, after='b0', label='b0.5')
a\b  b0  b0.5  b1  b2
a0    0    0   1   2
a1    3    1   4   5
```

If you want to target positions, you have to somehow specify the axis:

```
>>> a, b = arr1.axes
>>> # arr1.insert(42, before='b.i[1]', label='b0.5')
>>> arr1.insert(42, before=b.i[1], label='b0.5')
a\b  b0  b0.5  b1  b2
a0    0    42   1   2
a1    3    42   4   5
```

Insert an array which already has the axis

```
>>> arr3 = ndtest('a=a0,a1;b=b0.1,b0.2') + 42
>>> arr3
```

(continues on next page)

(continued from previous page)

```

a\b  b0.1  b0.2
a0    42    43
a1    44    45
>>> arr1.insert(arr3, before='b1')
a\b  b0  b0.1  b0.2  b1  b2
a0    0    42    43    1    2
a1    3    44    45    4    5

```

larray.Array.broadcast_with

Array.**broadcast_with**(*self*, *target*)

Returns an array that is (NumPy) broadcastable with target.

- all common axes must be either of length 1 or the same length
- extra axes in source can have any length and will be moved to the front
- extra axes in target can have any length and the result will have axes of length 1 for those axes

This is different from reshape which ensures the result has exactly the shape of the target.

Parameters

target [Array or collection of Axis]

Returns

Array

larray.Array.align

Array.**align**(*self*, *other*, *join*='outer', *fill_value*=nan, *axes*=None)

Align two arrays on their axes with the specified join method.

In other words, it ensure all common axes are compatible. Those arrays can then be used in binary operations.

Parameters

other [Array-like]

join [{ 'outer', 'inner', 'left', 'right', 'exact' }, optional]

Join method. For each axis common to both arrays:

- **outer:** will use a label if it is in either arrays axis (ordered like the first array).
This is the default as it results in no information loss.
- **inner:** will use a label if it is in both arrays axis (ordered like the first array).
- **left:** will use the first array axis labels.
- **right:** will use the other array axis labels.
- **exact:** instead of aligning, raise an error when axes to be aligned are not equal.

fill_value [scalar or Array, optional] Value used to fill cells corresponding to label combinations which are not common to both arrays. Defaults to NaN.

axes [AxisReference or sequence of them, optional] Axes to align. Need to be valid in both arrays. Defaults to None (all common axes). This must be specified when mixing anonymous and non-anonymous axes.

Returns

(left, right) [(Array, Array)] Aligned objects

Notes

Arrays with anonymous axes are currently not supported.

Examples

```
>>> arr1 = ndtest((2, 3))
>>> arr1
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
>>> arr2 = -ndtest((3, 2))
>>> # reorder array to make the test more interesting
>>> arr2 = arr2[['b1', 'b0']]
>>> arr2
a\b  b1  b0
a0  -1   0
a1  -3  -2
a2  -5  -4
```

Align arr1 and arr2

```
>>> aligned1, aligned2 = arr1.align(arr2)
>>> aligned1
a\b  b0  b1  b2
a0  0.0  1.0  2.0
a1  3.0  4.0  5.0
a2  nan  nan  nan
>>> aligned2
a\b  b0  b1  b2
a0  0.0 -1.0  nan
a1 -2.0 -3.0  nan
a2 -4.0 -5.0  nan
```

After aligning all common axes, one can then do operations between the two arrays

```
>>> aligned1 + aligned2
a\b  b0  b1  b2
a0  0.0  0.0  nan
a1  1.0  1.0  nan
a2  nan  nan  nan
```

Other kinds of joins are supported

```
>>> aligned1, aligned2 = arr1.align(arr2, join='inner')
>>> aligned1
a\b  b0  b1
a0  0.0  1.0
a1  3.0  4.0
>>> aligned2
a\b  b0  b1
a0  0.0 -1.0
```

(continues on next page)

(continued from previous page)

```

a1 -2.0 -3.0
>>> aligned1, aligned2 = arr1.align(arr2, join='left')
>>> aligned1
a\b  b0  b1  b2
a0  0.0  1.0  2.0
a1  3.0  4.0  5.0
>>> aligned2
a\b  b0  b1  b2
a0  0.0 -1.0 nan
a1 -2.0 -3.0 nan
>>> aligned1, aligned2 = arr1.align(arr2, join='right')
>>> aligned1
a\b  b1  b0
a0  1.0  0.0
a1  4.0  3.0
a2  nan  nan
>>> aligned2
a\b  b1  b0
a0 -1.0  0.0
a1 -3.0 -2.0
a2 -5.0 -4.0

```

The fill value for missing labels defaults to nan but can be changed to any compatible value.

```

>>> aligned1, aligned2 = arr1.align(arr2, fill_value=0)
>>> aligned1
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
a2   0   0   0
>>> aligned2
a\b  b0  b1  b2
a0   0  -1   0
a1  -2  -3   0
a2  -4  -5   0
>>> aligned1 + aligned2
a\b  b0  b1  b2
a0   0   0   2
a1   1   1   5
a2  -4  -5   0

```

It also works when either arrays (or both) have extra axes

```

>>> arr3 = ndtest((3, 2, 2))
>>> arr1
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> arr3
a  b\c  c0  c1
a0  b0   0   1
a0  b1   2   3
a1  b0   4   5
a1  b1   6   7
a2  b0   8   9
a2  b1  10  11
>>> aligned1, aligned2 = arr1.align(arr3, join='inner')

```

(continues on next page)

(continued from previous page)

```

>>> aligned1
a\b   b0   b1
a0  0.0  1.0
a1  3.0  4.0
>>> aligned2
a  b\c   c0   c1
a0 b0  0.0  1.0
a0 b1  2.0  3.0
a1 b0  4.0  5.0
a1 b1  6.0  7.0
>>> aligned1 + aligned2
a  b\c   c0   c1
a0 b0  0.0  1.0
a0 b1  3.0  4.0
a1 b0  7.0  8.0
a1 b1 10.0 11.0

```

One can also align only some specific axes (but in that case arrays might not be compatible)

```

>>> aligned1, aligned2 = arr1.align(arr2, axes='b')
>>> aligned1
a\b   b0   b1   b2
a0  0.0  1.0  2.0
a1  3.0  4.0  5.0
>>> aligned2
a\b   b0   b1   b2
a0  0.0 -1.0 nan
a1 -2.0 -3.0 nan
a2 -4.0 -5.0 nan

```

Test if two arrays are aligned

```

>>> arr1.align(arr2, join='exact')
Traceback (most recent call last):
...
ValueError: Both arrays are not aligned because align method with join='exact'
expected Axis(['a0', 'a1'], 'a') to be equal to Axis(['a0', 'a1', 'a2'], 'a')

```

Testing/Searching

<code>Array.equals(self, other[, rtol, atol, ...])</code>	Compares self with another array and returns True if they have the same axes and elements, False otherwise.
<code>Array.eq(self, other[, rtol, atol, nans_equal])</code>	Compares self with another array element-wise and returns an array of booleans.
<code>Array.isin(self, test_values[, ...])</code>	Computes whether each element of this array is in <i>test_values</i> .
<code>Array.nonzero(self)</code>	Returns the indices of the elements that are non-zero.
<code>Array.all(*axes_and_groups[, out, skipna, ...])</code>	Test whether all selected elements evaluate to True.
<code>Array.all_by(*axes_and_groups[, out, ...])</code>	Test whether all selected elements evaluate to True.
<code>Array.any(*axes_and_groups[, out, skipna, ...])</code>	Test whether any selected elements evaluate to True.
<code>Array.any_by(*axes_and_groups[, out, ...])</code>	Test whether any selected elements evaluate to True.
<code>Array.min(*axes_and_groups[, out, skipna, ...])</code>	Get minimum of array elements along given axes/groups.

Continued on next page

Table 38 – continued from previous page

<code>Array.min_by(*axes_and_groups[, out, ...])</code>	Get minimum of array elements for the given axes/groups.
<code>Array.max(*axes_and_groups[, out, skipna, ...])</code>	Get maximum of array elements along given axes/groups.
<code>Array.max_by(*axes_and_groups[, out, ...])</code>	Get maximum of array elements for the given axes/groups.
<code>Array.labelofmin(self[, axis])</code>	Returns labels of the minimum values along a given axis.
<code>Array.indexofmin(self[, axis])</code>	Returns indices of the minimum values along a given axis.
<code>Array.labelofmax(self[, axis])</code>	Returns labels of the maximum values along a given axis.
<code>Array.indexofmax(self[, axis])</code>	Returns indices of the maximum values along a given axis.

`larray.Array.equals`

`Array.equals` (*self*, *other*, *rtol*=0, *atol*=0, *nans_equal*=False, *check_axes*=False)

Compares self with another array and returns True if they have the same axes and elements, False otherwise.

Parameters

other [Array-like] Input array. `asarray()` is used on a non-Array input.

rtol [float or int, optional] The relative tolerance parameter (see Notes). Defaults to 0.

atol [float or int, optional] The absolute tolerance parameter (see Notes). Defaults to 0.

nans_equal [boolean, optional] Whether or not to consider NaN values at the same positions in the two arrays as equal. By default, an array containing NaN values is never equal to another array, even if that other array also contains NaN values at the same positions. The reason is that a NaN value is different from *anything*, including itself. Defaults to False.

check_axes [boolean, optional] Whether or not to check that the set of axes and their order is the same on both sides. Defaults to False. If False, two arrays with compatible axes (and the same data) will compare equal, even if some axis is missing on either side or if the axes are in a different order.

Returns

bool Returns True if self is equal to other.

See also:

`Array.eq`

Notes

For finite values, equals uses the following equation to test whether two values are equal:

$$\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$$

The above equation is not symmetric in array1 and array2, so that array1.equals(array2) might be different from array2.equals(array1) in some rare cases.

Examples

```
>>> arr1 = ndtest((2, 3))
>>> arr1
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> arr2 = arr1.copy()
>>> arr2.equals(arr1)
True
>>> arr2['b1'] += 1
>>> arr2.equals(arr1)
False
>>> arr3 = arr1.set_labels('a', ['x0', 'x1'])
>>> arr3.equals(arr1)
False
```

Test equality between two arrays within a given tolerance range. Return True if $\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$.

```
>>> arr1 = Array([6., 8.], "a=a0,a1")
>>> arr1
a  a0  a1
   6.0 8.0
>>> arr2 = Array([5.999, 8.001], "a=a0,a1")
>>> arr2
a  a0  a1
   5.999 8.001
>>> arr2.equals(arr1)
False
>>> arr2.equals(arr1, atol=0.01)
True
>>> arr2.equals(arr1, rtol=0.01)
True
```

Arrays with NaN values

```
>>> arr1 = ndtest((2, 3), dtype=float)
>>> arr1['a1', 'b1'] = nan
>>> arr1
a\b  b0  b1  b2
a0   0.0 1.0 2.0
a1   3.0 nan 5.0
>>> arr2 = arr1.copy()
>>> # By default, an array containing NaN values is never equal to another array,
>>> # even if that other array also contains NaN values at the same positions.
>>> # The reason is that a NaN value is different from *anything*, including
↳ itself.
```

(continues on next page)

(continued from previous page)

```
>>> arr2.equals(arr1)
False
>>> # set flag nans_equal to True to overwrite this behavior
>>> arr2.equals(arr1, nans_equal=True)
True
```

Arrays with the same data but different axes

```
>>> arr1 = ndtest((2, 2))
>>> arr1
a\b  b0  b1
a0   0   1
a1   2   3
>>> arr2 = arr1.transpose()
>>> arr2
b\a  a0  a1
b0   0   2
b1   1   3
>>> arr2.equals(arr1)
True
>>> arr2.equals(arr1, check_axes=True)
False
>>> arr2 = arr1.expand('c=c0,c1')
>>> arr2
a  b\c  c0  c1
a0  b0   0   0
a0  b1   1   1
a1  b0   2   2
a1  b1   3   3
>>> arr2.equals(arr1)
True
>>> arr2.equals(arr1, check_axes=True)
False
```

larray.Array.eq

`Array.eq(self, other, rtol=0, atol=0, nans_equal=False)`

Compares self with another array element-wise and returns an array of booleans.

Parameters

other [Array-like] Input array. `asarray()` is used on a non-Array input.

rtol [float or int, optional] The relative tolerance parameter (see Notes). Defaults to 0.

atol [float or int, optional] The absolute tolerance parameter (see Notes). Defaults to 0.

nans_equal [boolean, optional] Whether or not to consider Nan values at the same positions in the two arrays as equal. By default, an array containing NaN values is never equal to another array, even if that other array also contains NaN values at the same positions. The reason is that a NaN value is different from *anything*, including itself. Defaults to False.

Returns

Array Boolean array where each cell tells whether corresponding elements of self and other are equal within a tolerance range if given. If `nans_equal=True`, corresponding elements with NaN values will be considered as equal.

See also:

[*Array.equals*](#)

Notes

For finite values, eq uses the following equation to test whether two values are equal:

$$\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$$

The above equation is not symmetric in array1 and array2, so that array1.eq(array2) might be different from array2.eq(array1) in some rare cases.

Examples

```
>>> arr1 = Array([6., np.nan, 8.], "a=a0..a2")
>>> arr1
a   a0   a1   a2
   6.0 nan  8.0
```

Default behavior (same as == operator)

```
>>> arr1.eq(arr1)
a   a0   a1   a2
   True False True
```

Test equality between two arrays within a given tolerance range. Return True if $\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$.

```
>>> arr2 = Array([5.999, np.nan, 8.001], "a=a0..a2")
>>> arr2
a   a0   a1   a2
   5.999 nan  8.001
>>> arr1.eq(arr2, nans_equal=True)
a   a0   a1   a2
   False True False
>>> arr1.eq(arr2, atol=0.01, nans_equal=True)
a   a0   a1   a2
   True True True
>>> arr1.eq(arr2, rtol=0.01, nans_equal=True)
a   a0   a1   a2
   True True True
```

larray.Array.isin

`Array.isin(self, test_values, assume_unique=False, invert=False)`

Computes whether each element of this array is in *test_values*. Returns a boolean array of the same shape as this array that is True where the array element is in *test_values* and False otherwise.

Parameters

test_values [array_like or set] The values against which to test each element of this array. If *test_values* is not a 1D array, it will be converted to one.

assume_unique [bool, optional] If True, this array and *test_values* are both assumed to be unique, which can speed up the calculation. Defaults to False.

invert [bool, optional] If True, the values in the returned array are inverted, as if calculating *element not in test_values*. Defaults to False. `isin(a, b, invert=True)` is equivalent to (but faster than) `~isin(a, b)`.

Returns

Array boolean array of the same shape as this array that is True where the array element is in *test_values* and False otherwise.

Examples

```
>>> arr = ndtest((2, 3))
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> arr.isin([1, 5, 7])
a\b    b0    b1    b2
a0  False  True  False
a1  False  False  True
>>> arr[arr.isin([1, 5, 7])]
a_b  a0_b1  a1_b2
      1      5
```

larray.Array.nonzero

`Array.nonzero(self)`

Returns the indices of the elements that are non-zero.

Specifically, it returns a tuple of arrays (one for each dimension) containing the indices of the non-zero elements in that dimension.

Returns

tuple of arrays [tuple] Indices of elements that are non-zero.

Examples

```
>>> arr = ndtest((2, 3))
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> cond = arr > 1
>>> cond
a\b    b0    b1    b2
a0  False  False  True
a1  True   True   True
>>> a, b = cond.nonzero()
>>> a
a.i[a_b  a0_b2  a1_b0  a1_b1  a1_b2
      0      1      1      1]
```

(continues on next page)

(continued from previous page)

```

>>> b
b.i[a_b  a0_b2  a1_b0  a1_b1  a1_b2
      2      0      1      2]
>>> # equivalent to arr[cond]
>>> arr[cond.nonzero()]
a_b  a0_b2  a1_b0  a1_b1  a1_b2
      2      3      4      5

```

larray.Array.all

`Array.all(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Test whether all selected elements evaluate to True.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those]
Axis(es) or group(s) along which the AND reduction is performed. The default (no axis or group) is to perform the AND reduction over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the AND reduction over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns**Array of bool or bool****See also:***Array.all_by, Array.any, Array.any_by***Examples**

```

>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
a2    8   9  10  11
a3   12  13  14  15
>>> barr = arr < 6
>>> barr
a\b      b0      b1      b2      b3
a0   True    True    True    True
a1   True    True   False   False
a2  False   False   False   False
a3  False   False   False   False
>>> barr.all()
False
>>> # along axis 'a'
>>> barr.all('a')
b      b0      b1      b2      b3
      False   False   False   False
>>> # along axis 'b'
>>> barr.all('b')
a      a0      a1      a2      a3
      True   False   False   False

```

Select some rows only

```

>>> barr.all(['a0', 'a1'])
b      b0      b1      b2      b3
      True    True   False   False
>>> # or equivalently
>>> # barr.all('a0,a1')

```

Split an axis in several parts

```

>>> barr.all([( 'a0', 'a1'], [ 'a2', 'a3']))
a\b      b0      b1      b2      b3
a0,a1   True    True   False   False
a2,a3  False   False   False   False
>>> # or equivalently
>>> # barr.all('a0,a1;a2,a3')

```

Same with renaming

```

>>> barr.all((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b      b0      b1      b2      b3
a01   True    True   False   False

```

(continues on next page)

(continued from previous page)

```
a23 False False False False
>>> # or equivalently
>>> # barr.all('a0,a1>>a01;a2,a3>>a23')
```

`larray.Array.all_by`

`Array.all_by(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`
Test whether all selected elements evaluate to True.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The AND reduction is performed along all axes except the given one(s). For groups, AND reduction is performed along groups and non associated axes. The default (no axis or group) is to perform the AND reduction over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the AND reduction over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array of bool or bool

See also:

Array.all, *Array.any*, *Array.any_by*

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> barr = arr < 6
>>> barr
a\b    b0    b1    b2    b3
a0  True   True   True   True
a1  True   True  False  False
a2  False  False  False  False
a3  False  False  False  False
>>> barr.all_by()
False
>>> # by axis 'a'
>>> barr.all_by('a')
a  a0  a1  a2  a3
   True False False False
>>> # by axis 'b'
>>> barr.all_by('b')
b  b0  b1  b2  b3
   False False False False
```

Select some rows only

```
>>> barr.all_by(['a0', 'a1'])
False
>>> # or equivalently
>>> # barr.all_by('a0,a1')
```

Split an axis in several parts

```
>>> barr.all_by(['a0', 'a1'], ['a2', 'a3'])
a  a0,a1  a2,a3
   False  False
>>> # or equivalently
>>> # barr.all_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> barr.all_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01  a23
   False False
>>> # or equivalently
>>> # barr.all_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.any

`Array.any(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Test whether any selected elements evaluate to True.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those]
Axis(es) or group(s) along which the OR reduction is performed. The default (no axis or group) is to perform the OR reduction over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the OR reduction over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array of bool or bool

See also:

[`Array.any_by`](#), [`Array.all`](#), [`Array.all_by`](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
a2    8   9  10  11
a3   12  13  14  15
>>> barr = arr < 6
>>> barr
a\b    b0    b1    b2    b3
a0   True   True   True   True
a1   True   True  False  False
a2  False  False  False  False
a3  False  False  False  False
>>> barr.any()
True
>>> # along axis 'a'
>>> barr.any('a')
b    b0    b1    b2    b3
    True   True   True   True
>>> # along axis 'b'
>>> barr.any('b')
a    a0    a1    a2    a3
    True   True  False  False
```

Select some rows only

```
>>> barr.any(['a0', 'a1'])
b    b0    b1    b2    b3
    True   True   True   True
>>> # or equivalently
>>> # barr.any('a0,a1')
```

Split an axis in several parts

```
>>> barr.any(['a0', 'a1'], ['a2', 'a3'])
a\b    b0    b1    b2    b3
a0,a1  True   True   True   True
a2,a3  False  False  False  False
>>> # or equivalently
>>> # barr.any('a0,a1;a2,a3')
```

Same with renaming

```
>>> barr.any((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b    b0    b1    b2    b3
a01  True   True   True   True
a23  False  False  False  False
>>> # or equivalently
>>> # barr.any('a0,a1>>a01;a2,a3>>a23')
```

`larray.Array.any_by`

`Array.any_by(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Test whether any selected elements evaluate to True.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The OR reduction is performed along all axes except the given one(s). For groups, OR reduction is performed along groups and non associated axes. The default (no axis or group) is to perform the OR reduction over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to perform the OR reduction over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator ' >> ' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. `keepaxes='summation'`). Defaults to False.

Returns

Array of bool or bool

See also:

[`Array.any`](#), [`Array.all`](#), [`Array.all_by`](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> barr = arr < 6
>>> barr
a\b      b0      b1      b2      b3
a0   True   True   True   True
a1   True   True  False  False
a2  False  False  False  False
a3  False  False  False  False
>>> barr.any_by()
True
>>> # by axis 'a'
>>> barr.any_by('a')
a   a0   a1   a2   a3
   True  True False False
>>> # by axis 'b'
>>> barr.any_by('b')
b   b0   b1   b2   b3
   True  True  True  True
```

Select some rows only

```
>>> barr.any_by(['a0', 'a1'])
True
>>> # or equivalently
>>> # barr.any_by('a0,a1')
```

Split an axis in several parts

```
>>> barr.any_by((['a0', 'a1'], ['a2', 'a3']))
a  a0,a1  a2,a3
   True  False
>>> # or equivalently
>>> # barr.any_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> barr.any_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a   a01   a23
   True  False
>>> # or equivalently
>>> # barr.any_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.min

`Array.min(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Get minimum of array elements along given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those]
Axis(es) or group(s) along which the minimum is searched. The default (no axis or group) is to search the minimum over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to search the minimum over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator ' >> ' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[`Array.min_by`](#), [`Array.max`](#), [`Array.max_by`](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
a2    8   9  10  11
a3   12  13  14  15
>>> arr.min()
0
>>> # along axis 'a'
>>> arr.min('a')
b  b0  b1  b2  b3
   0   1   2   3
>>> # along axis 'b'
>>> arr.min('b')
a  a0  a1  a2  a3
   0   4   8  12
```

Select some rows only

```
>>> arr.min(['a0', 'a1'])
b  b0  b1  b2  b3
   0   1   2   3
>>> # or equivalently
>>> # arr.min('a0,a1')
```

Split an axis in several parts

```
>>> arr.min(['a0', 'a1'], ['a2', 'a3'])
a\b  b0  b1  b2  b3
a0,a1  0   1   2   3
a2,a3   8   9  10  11
>>> # or equivalently
>>> # arr.min('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.min((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b  b0  b1  b2  b3
a01   0   1   2   3
a23   8   9  10  11
>>> # or equivalently
>>> # arr.min('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.min_by

`Array.min_by(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Get minimum of array elements for the given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The minimum is searched along all axes except the given one(s). For groups, minimum is searched along groups and non associated axes. The default (no axis or group) is to search the minimum over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to search the minimum over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[*Array.min*](#), [*Array.max*](#), [*Array.max_by*](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
a2    8   9  10  11
a3   12  13  14  15
>>> arr.min_by()
0
```

(continues on next page)

(continued from previous page)

```
>>> # along axis 'a'
>>> arr.min_by('a')
a  a0  a1  a2  a3
   0   4   8  12
>>> # along axis 'b'
>>> arr.min_by('b')
b  b0  b1  b2  b3
   0   1   2   3
```

Select some rows only

```
>>> arr.min_by(['a0', 'a1'])
0
>>> # or equivalently
>>> # arr.min_by('a0,a1')
```

Split an axis in several parts

```
>>> arr.min_by(['a0', 'a1'], ['a2', 'a3'])
a  a0,a1  a2,a3
   0       8
>>> # or equivalently
>>> # arr.min_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.min_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01  a23
   0    8
>>> # or equivalently
>>> # arr.min_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.max

`Array.max(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Get maximum of array elements along given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those]
Axis(es) or group(s) along which the maximum is searched. The default (no axis or group) is to search the maximum over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to search the maximum over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string

- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')
- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator '>>' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[*Array.max_by*](#), [*Array.min*](#), [*Array.min_by*](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.max()
15
>>> # along axis 'a'
>>> arr.max('a')
b  b0  b1  b2  b3
   12  13  14  15
>>> # along axis 'b'
>>> arr.max('b')
a  a0  a1  a2  a3
   3   7  11  15
```

Select some rows only

```
>>> arr.max(['a0', 'a1'])
b  b0  b1  b2  b3
   4   5   6   7
```

(continues on next page)

(continued from previous page)

```
>>> # or equivalently
>>> # arr.max('a0,a1')
```

Split an axis in several parts

```
>>> arr.max(['a0', 'a1'], ['a2', 'a3'])
a\b  b0  b1  b2  b3
a0,a1  4   5   6   7
a2,a3 12  13  14  15
>>> # or equivalently
>>> # arr.max('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.max((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a\b  b0  b1  b2  b3
a01   4   5   6   7
a23  12  13  14  15
>>> # or equivalently
>>> # arr.max('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.max_by

`Array.max_by(*axes_and_groups, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Get maximum of array elements for the given axes/groups.

Parameters

***axes_and_groups** [None or int or str or Axis or Group or any combination of those] The maximum is searched along all axes except the given one(s). For groups, maximum is searched along groups and non associated axes. The default (no axis or group) is to search the maximum over all the dimensions of the input array.

An axis can be referred by:

- its index (integer). Index can be a negative integer, in which case it counts from the last to the first axis.
- its name (str or AxisReference). You can use either a simple string ('axis_name') or the special variable X (X.axis_name).
- a variable (Axis). If the axis has been defined previously and assigned to a variable, you can pass it as argument.

You may not want to search the maximum over a whole axis but over a selection of specific labels. To do so, you have several possibilities:

- (['a1', 'a3', 'a5'], 'b1, b3, b5') : labels separated by commas in a list or a string
- ('a1:a5:2') : select labels using a slice (general syntax is 'start:end:step' where 'step' is optional and 1 by default).
- (a='a1, a2, a3', X.b['b1, b2, b3']) : in case of possible ambiguity, i.e. if labels can belong to more than one axis, you must precise the axis.
- ('a1:a3; a5:a7', b='b0,b2; b1,b3') : create several groups with semicolons. Names are simply given by the concatenation of labels (here: 'a1,a2,a3', 'a5,a6,a7', 'b0,b2' and 'b1,b3')

- ('a1:a3 >> a123', 'b[b0,b2] >> b12') : operator ' >> ' allows to rename groups.

out [Array, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0's and 1.0's). Axes and labels can be different, only the shape matters. Defaults to None (create a new array).

skipna [bool, optional] Whether or not to skip NaN (null) values. If False, resulting cells will be NaN if any of the aggregated cells is NaN. Defaults to True.

keepaxes [bool or label-like, optional] Whether or not reduced axes are left in the result as dimensions with size one. If True, reduced axes will contain a unique label representing the applied aggregation (e.g. 'sum', 'prod', ...). It is possible to override this label by passing a specific value (e.g. keepaxes='summation'). Defaults to False.

Returns

Array or scalar

See also:

[*Array.max*](#), [*Array.min*](#), [*Array.min_by*](#)

Examples

```
>>> arr = ndtest((4, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
a3  12  13  14  15
>>> arr.max_by()
15
>>> # along axis 'a'
>>> arr.max_by('a')
a  a0  a1  a2  a3
   3   7  11  15
>>> # along axis 'b'
>>> arr.max_by('b')
b  b0  b1  b2  b3
   12  13  14  15
```

Select some rows only

```
>>> arr.max_by(['a0', 'a1'])
7
>>> # or equivalently
>>> # arr.max_by('a0,a1')
```

Split an axis in several parts

```
>>> arr.max_by(['a0', 'a1'], ['a2', 'a3'])
a  a0,a1  a2,a3
   7      15
>>> # or equivalently
>>> # arr.max_by('a0,a1;a2,a3')
```

Same with renaming

```
>>> arr.max_by((X.a['a0', 'a1'] >> 'a01', X.a['a2', 'a3'] >> 'a23'))
a  a01  a23
   7    15
>>> # or equivalently
>>> # arr.max_by('a0,a1>>a01;a2,a3>>a23')
```

larray.Array.labelofmin

`Array.labelofmin(self, axis=None)`

Returns labels of the minimum values along a given axis.

Parameters

axis [int or str or Axis, optional] Axis along which to work. If not specified, works on the full array.

Returns

Array

Notes

In case of multiple occurrences of the minimum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> nat = Axis('nat=BE,FR,IT')
>>> sex = Axis('sex=M,F')
>>> arr = Array([[0, 1], [3, 2], [2, 5]], [nat, sex])
>>> arr
nat\sex  M  F
      BE  0  1
      FR  3  2
      IT  2  5
>>> arr.labelofmin('sex')
nat  BE  FR  IT
      M   F   M
>>> arr.labelofmin()
('BE', 'M')
```

larray.Array.indexofmin

`Array.indexofmin(self, axis=None)`

Returns indices of the minimum values along a given axis.

Parameters

axis [int or str or Axis, optional] Axis along which to work. If not specified, works on the full array.

Returns

Array

Notes

In case of multiple occurrences of the minimum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> nat = Axis('nat=BE,FR,IT')
>>> sex = Axis('sex=M,F')
>>> arr = Array([[0, 1], [3, 2], [2, 5]], [nat, sex])
>>> arr
nat\sex  M  F
      BE  0  1
      FR  3  2
      IT  2  5
>>> arr.indexofmin('sex')
nat  BE  FR  IT
     0   1   0
>>> arr.indexofmin()
(0, 0)
```

larray.Array.labelofmax

`Array.labelofmax(self, axis=None)`

Returns labels of the maximum values along a given axis.

Parameters

axis [int or str or Axis, optional] Axis along which to work. If not specified, works on the full array.

Returns

Array

Notes

In case of multiple occurrences of the maximum values, the labels corresponding to the first occurrence are returned.

Examples

```
>>> nat = Axis('nat=BE,FR,IT')
>>> sex = Axis('sex=M,F')
>>> arr = Array([[0, 1], [3, 2], [2, 5]], [nat, sex])
>>> arr
nat\sex  M  F
      BE  0  1
      FR  3  2
      IT  2  5
>>> arr.labelofmax('sex')
nat  BE  FR  IT
     F   M   F
```

(continues on next page)

(continued from previous page)

```
>>> arr.labelofmax()
('IT', 'F')
```

larray.Array.indexofmax

`Array.indexofmax(self, axis=None)`

Returns indices of the maximum values along a given axis.

Parameters

axis [int or str or Axis, optional] Axis along which to work. If not specified, works on the full array.

Returns

Array

Notes

In case of multiple occurrences of the maximum values, the labels corresponding to the first occurrence are returned.

Examples

```
>>> nat = Axis('nat=BE,FR,IT')
>>> sex = Axis('sex=M,F')
>>> arr = Array([[0, 1], [3, 2], [2, 5]], [nat, sex])
>>> arr
nat\sex  M  F
      BE  0  1
      FR  3  2
      IT  2  5
>>> arr.indexofmax('sex')
nat  BE  FR  IT
     1   0   1
>>> arr.indexofmax()
(2, 1)
```

Iterating

<code>Array.keys(self[, axes, ascending])</code>	Returns a view on the array labels along axes.
<code>Array.values(self[, axes, ascending])</code>	Returns a view on the values of the array along axes.
<code>Array.items(self[, axes, ascending])</code>	Returns a (label, value) view of the array along axes.

larray.Array.keys

`Array.keys` (*self*, *axes=None*, *ascending=True*)

Returns a view on the array labels along axes.

Parameters

axes [int, str or Axis or tuple of them, optional] Axis or axes along which to iterate and in which order. Defaults to None (all axes in the order they are in the array).

ascending [bool, optional] Whether or not to iterate the axes in ascending order (from start to end). Defaults to True.

Returns

Sequence An object you can iterate (loop) on and index by position to get the Nth label along axes.

Examples

First, define a small helper function to make the following examples more readable.

```
>>> def str_key(key):  
...     return tuple(str(k) for k in key)
```

Then create a test array:

```
>>> arr = ndtest((2, 2))  
>>> arr  
a\b  b0  b1  
a0    0   1  
a1    2   3
```

By default it iterates on all axes, in the order they are in the array.

```
>>> for key in arr.keys():  
...     # print both the actual key object, and a (nicer) string representation  
...     print(key, "->", str_key(key))  
(a.i[0], b.i[0]) -> ('a0', 'b0')  
(a.i[0], b.i[1]) -> ('a0', 'b1')  
(a.i[1], b.i[0]) -> ('a1', 'b0')  
(a.i[1], b.i[1]) -> ('a1', 'b1')  
>>> for key in arr.keys(ascending=False):  
...     print(str_key(key))  
('a1', 'b1')  
('a1', 'b0')  
('a0', 'b1')  
('a0', 'b0')
```

but you can specify another axis order:

```
>>> for key in arr.keys(('b', 'a')):  
...     print(str_key(key))  
('b0', 'a0')  
('b0', 'a1')  
('b1', 'a0')  
('b1', 'a1')
```


One can specify less axes than the array has:

```
>>> # iterate on the "b" axis, that is return each label along the "b" axis
... for key in arr.keys('b'):
...     print(str_key(key))
('b0',)
('b1',)
```

One can also access elements of the key sequence directly, instead of iterating over it. Say we want to retrieve the first and last keys of our array, we could write:

```
>>> keys = arr.keys()
>>> first_key = keys[0]
>>> str_key(first_key)
('a0', 'b0')
>>> last_key = keys[-1]
>>> str_key(last_key)
('a1', 'b1')
```

larray.Array.values

Array.**values** (*self*, *axes=None*, *ascending=True*)

Returns a view on the values of the array along axes.

Parameters

axes [int, str or Axis or tuple of them, optional] Axis or axes along which to iterate and in which order. Defaults to None (all axes in the order they are in the array).

ascending [bool, optional] Whether or not to iterate the axes in ascending order (from start to end). Defaults to True.

Returns

Sequence An object you can iterate (loop) on and index by position.

Examples

```
>>> arr = ndtest((2, 2))
>>> arr
a\b  b0  b1
a0   0   1
a1   2   3
```

By default it iterates on all axes, in the order they are in the array.

```
>>> for value in arr.values():
...     print(value)
0
1
2
3
>>> for value in arr.values(ascending=False):
...     print(value)
3
2
```

(continues on next page)

(continued from previous page)

```
1
0
```

but you can specify another axis order:

```
>>> for value in arr.values(('b', 'a')):
...     print(value)
0
2
1
3
```

When you specify less axes than the array has, you get arrays back:

```
>>> # iterate on the "b" axis, that is return the (sub)array for each label along
↳the "b" axis
... for value in arr.values('b'):
...     print(value)
a  a0  a1
   0   2
a  a0  a1
   1   3
>>> # iterate on the "b" axis, that is return the (sub)array for each label along
↳the "b" axis
... for value in arr.values('b', ascending=False):
...     print(value)
a  a0  a1
   1   3
a  a0  a1
   0   2
```

One can also access elements of the value sequence directly, instead of iterating over it. Say we want to retrieve the first and last values of our array, we could write:

```
>>> values = arr.values()
>>> values[0]
0
>>> values[-1]
3
```

larray.Array.items

`Array.items` (*self*, *axes=None*, *ascending=True*)

Returns a (label, value) view of the array along axes.

Parameters

axes [int, str or Axis or tuple of them, optional] Axis or axes along which to iterate and in which order. Defaults to None (all axes in the order they are in the array).

ascending [bool, optional] Whether or not to iterate the axes in ascending order (from start to end). Defaults to True.

Returns

Sequence An object you can iterate (loop) on and index by position to get the Nth (label, value) couple along axes.

Examples

First, define a small helper function to make the following examples more readable.

```
>>> def str_key(key):
...     return tuple(str(k) for k in key)
```

Then create a test array:

```
>>> arr = ndtest((2, 2))
>>> arr
a\b  b0  b1
a0   0   1
a1   2   3
```

By default it iterates on all axes, in the order they are in the array.

```
>>> for key, value in arr.items():
...     print(str_key(key), "->", value)
('a0', 'b0') -> 0
('a0', 'b1') -> 1
('a1', 'b0') -> 2
('a1', 'b1') -> 3
>>> for key, value in arr.items(ascending=False):
...     print(str_key(key), "->", value)
('a1', 'b1') -> 3
('a1', 'b0') -> 2
('a0', 'b1') -> 1
('a0', 'b0') -> 0
```

but you can specify another axis order:

```
>>> for key, value in arr.items(('b', 'a')):
...     print(str_key(key), "->", value)
('b0', 'a0') -> 0
('b0', 'a1') -> 2
('b1', 'a0') -> 1
('b1', 'a1') -> 3
```

When you specify less axes than the array has, you get arrays back:

```
>>> # iterate on the "b" axis, that is return the (sub)array for each label along
↳the "b" axis
... for key, value in arr.items('b'):
...     print(str_key(key), value, sep="\n")
('b0',)
a  a0  a1
   0   2
('b1',)
a  a0  a1
   1   3
```

One can also access elements of the items sequence directly, instead of iterating over it. Say we want to retrieve the first and last key-value pairs of our array, we could write:

```
>>> items = arr.items()
>>> first_key, first_value = items[0]
```

(continues on next page)

(continued from previous page)

```

>>> str_key(first_key)
('a0', 'b0')
>>> first_value
0
>>> last_key, last_value = items[-1]
>>> str_key(last_key)
('a1', 'b1')
>>> last_value
3

```

Operators

@	Matrix multiplication
---	-----------------------

Miscellaneous

<code>Array.divnot0(self, other)</code>	Divides array by other, but returns 0.0 where other is 0.
<code>Array.clip(self[, minval, maxval, out])</code>	Clip (limit) the values in an array.
<code>Array.shift(self, axis[, n])</code>	Shifts the cells of the array n-times to the right along axis.
<code>Array.roll(self[, axis, n])</code>	Rolls the cells of the array n-times to the right along axis.
<code>Array.diff(self[, axis, d, n, label])</code>	Calculates the n-th order discrete difference along a given axis.
<code>Array.unique(self[, axes, sort, sep])</code>	Returns unique values (optionally along axes)
<code>Array.to_clipboard(self, *args, **kwargs)</code>	Sends the content of the array to clipboard.

larray.Array.divnot0

`Array.divnot0(self, other)`

Divides array by other, but returns 0.0 where other is 0.

Parameters

other [scalar or Array] What to divide by.

Returns

Array Array divided by other, 0.0 where other is 0

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> a = ndtest((nat, sex))
>>> a
nat\sex  M  F
      BE  0  1
      FO  2  3
>>> b = ndtest(sex)
>>> b
sex  M  F
     0  1
>>> a / b
nat\sex  M      F
      BE  nan  1.0
      FO  inf  3.0
>>> a.divnot0(b)
nat\sex  M      F
      BE  0.0  1.0
      FO  0.0  3.0
```

larray.Array.clip

`Array.clip(self, minval=None, maxval=None, out=None)`

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval bounds. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters

minval [scalar or array-like, optional] Minimum value. If `None`, clipping is not performed on lower bound. Defaults to `None`.

maxval [scalar or array-like, optional] Maximum value. If `None`, clipping is not performed on upper bound. Defaults to `None`.

out [Array, optional] The results will be placed in this array.

Returns

Array An array with the elements of the current array, but where values $< minval$ are replaced with `minval`, and those $> maxval$ with `maxval`.

Notes

- At least either `minval` or `maxval` must be defined.
- If `minval` and/or `maxval` are array_like, broadcast will occur between `self`, `minval` and `maxval`.

Examples

```
>>> arr = ndtest((3, 3)) - 3
>>> arr
a\b  b0  b1  b2
a0   -3  -2  -1
a1    0   1   2
a2    3   4   5
>>> arr.clip(0, 2)
a\b  b0  b1  b2
a0    0   0   0
a1    0   1   2
a2    2   2   2
```

Clipping on lower bound only

```
>>> arr.clip(0)
a\b  b0  b1  b2
a0    0   0   0
a1    0   1   2
a2    3   4   5
```

Clipping on upper bound only

```
>>> arr.clip(maxval=2)
a\b  b0  b1  b2
a0   -3  -2  -1
a1    0   1   2
a2    2   2   2
```

`larray.Array.shift`

`Array.shift` (*self*, *axis*, *n=1*)

Shifts the cells of the array *n*-times to the right along axis.

Parameters

axis [int, str or Axis] Axis for which we want to perform the shift.

n [int, optional] Number of cells to shift. Defaults to 1.

Returns

Array

See also:

[`Array.roll`](#) cells which are pushed “outside of the axis” are reintroduced on the opposite side of the axis instead of being dropped.

Examples

```
>>> arr = ndtest('sex=M,F;year=2019..2021')
>>> arr
sex\year  2019  2020  2021
      M      0      1      2
      F      3      4      5
>>> arr.shift('year')
sex\year  2020  2021
      M      0      1
      F      3      4
>>> arr.shift('year', n=-1)
sex\year  2019  2020
      M      1      2
      F      4      5
```

larray.Array.roll

`Array.roll(self, axis=None, n=1)`

Rolls the cells of the array n-times to the right along axis. Cells which would be pushed “outside of the axis” are reintroduced on the opposite side of the axis.

Parameters

axis [int, str or Axis, optional] Axis along which to roll. Defaults to None (all axes).

n [int or Array, optional] Number of positions to roll. Defaults to 1. Use a negative integers to roll left. If n is an Array the number of positions rolled can vary along the axes of n.

Returns

Array

See also:

Array.shift cells which are pushed “outside of the axis” are dropped instead of being reintroduced on the opposite side of the axis.

Examples

```
>>> arr = ndtest('sex=M,F;year=2019..2021')
>>> arr
sex\year  2019  2020  2021
      M      0      1      2
      F      3      4      5
>>> arr.roll('year')
sex\year  2019  2020  2021
      M      2      0      1
      F      5      3      4
```

One can also roll by a different amount depending on another axis

```
>>> # let us roll by 1 for men and by 2 for women
>>> n = sequence(arr.sex, initial=1)
>>> n
sex  M  F
```

(continues on next page)

(continued from previous page)

```

1 2
>>> arr.roll('year', n)
sex\year  2019  2020  2021
      M      2      0      1
      F      4      5      3

```

larray.Array.diff

`Array.diff` (*self*, *axis=-1*, *d=1*, *n=1*, *label='upper'*)

Calculates the *n*-th order discrete difference along a 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, str, Group or Axis, optional] Axis or group along which the difference is taken. Defaults to the last axis.

d [int, optional] Periods to shift for forming difference. Defaults to 1.

n [int, optional] The number of times values are differenced. Defaults to 1.

label [{‘lower’, ‘upper’}, optional] The new labels in *axis* will have the labels of either the array being subtracted (‘lower’) or the array it is subtracted from (‘upper’). Defaults to ‘upper’.

Returns

Array The *n*-th order differences. The shape of the output is the same as *a* except for *axis* which is smaller by $n * d$.

Examples

```

>>> a = ndtest('sex=M,F;type=type1,type2,type3').cumsum('type')
>>> a
sex\type  type1  type2  type3
      M      0      1      3
      F      3      7     12
>>> a.diff()
sex\type  type2  type3
      M      1      2
      F      4      5
>>> a.diff(n=2)
sex\type  type3
      M      1
      F      1
>>> a.diff('sex')
sex\type  type1  type2  type3
      F      3      6      9
>>> a.diff(a.type['type2':])
sex\type  type3
      M      2
      F      5

```


larray.Array.unique

`Array.unique` (*self*, *axes=None*, *sort=False*, *sep='_'*)

Returns unique values (optionally along axes)

Parameters

axes [axis reference (int, str, Axis) or sequence of them, optional] Axis or axes along which to compute unique values. Defaults to None (all axes).

sort [bool, optional] Whether or not to sort unique values. Defaults to False. Sorting is not implemented yet for `unique()` along multiple axes.

sep [str, optional] Separator when several labels need to be combined. Defaults to `'_'`.

Returns

Array array with unique values

Examples

```
>>> arr = Array([[0, 2, 0, 0],
...             [1, 1, 1, 0]], 'a=a0,a1;b=b0..b3')
>>> arr
a\b  b0  b1  b2  b3
a0    0   2   0   0
a1    1   1   1   0
```

By default `unique()` returns the first occurrence of each unique value in the order it appears:

```
>>> arr.unique()
a_b  a0_b0  a0_b1  a1_b0
      0      2      1
```

To sort the unique values, use the `sort` argument:

```
>>> arr.unique(sort=True)
a_b  a0_b0  a1_b0  a0_b1
      0      1      2
```

One can also compute unique sub-arrays (i.e. combination of values) along axes. In our example the `a0=0, a1=1` combination appears twice along the `'b'` axis, so `'b2'` is not returned:

```
>>> arr.unique('b')
a\b  b0  b1  b3
a0    0   2   0
a1    1   1   0
>>> arr.unique('b', sort=True)
a\b  b3  b0  b1
a0    0   0   2
a1    0   1   1
```

larray.Array.to_clipboard

`Array.to_clipboard(self, *args, **kwargs)`

Sends the content of the array to clipboard.

Using `to_clipboard()` makes it possible to paste the content of the array into a file (Excel, ascii file,...).

Examples

```
>>> a = ndtest('nat=BE,F0;sex=M,F')
>>> a.to_clipboard()
```

Converting to Pandas objects

<code>Array.to_series(self[, name, dropna])</code>	Converts an Array into a Pandas Series.
<code>Array.to_frame(self[, fold_last_axis_name, ...])</code>	Converts an Array into a Pandas DataFrame.

larray.Array.to_series

`Array.to_series(self, name=None, dropna=False)`

Converts an Array into a Pandas Series.

Parameters

name [str, optional] Name of the series. Defaults to None.

dropna [bool, optional.] False by default.

Returns

Pandas Series

Notes

Since pandas does not provide a way to handle metadata (yet), all metadata associated with the array will be lost.

Examples

```
>>> arr = ndtest((2, 3), dtype=float)
>>> arr
a\b   b0   b1   b2
a0  0.0  1.0  2.0
a1  3.0  4.0  5.0
>>> arr.to_series()
a      b
a0  b0    0.0
     b1    1.0
     b2    2.0
a1  b0    3.0
     b1    4.0
```

(continues on next page)

(continued from previous page)

```

    b2    5.0
dtype: float64

```

Set a name

```

>>> arr.to_series('my_name')
      a    b
a0  b0    0.0
    b1    1.0
    b2    2.0
a1  b0    3.0
    b1    4.0
    b2    5.0
Name: my_name, dtype: float64

```

Drop NaN values

```

>>> arr['b1'] = nan
>>> arr
a\b   b0   b1   b2
a0  0.0  nan  2.0
a1  3.0  nan  5.0
>>> arr.to_series(dropna=True)
a    b
a0  b0    0.0
    b2    2.0
a1  b0    3.0
    b2    5.0
dtype: float64

```

larray.Array.to_frame

`Array.to_frame(self, fold_last_axis_name=False, dropna=None)`
 Converts an Array into a Pandas DataFrame.

Parameters

fold_last_axis_name [bool, optional] Defaults to False.

dropna [{ 'any', 'all', None }, optional]

- any : if any NA values are present, drop that label
- all : if all values are NA, drop that label
- None by default.

Returns

Pandas DataFrame

Notes

Since pandas does not provide a way to handle metadata (yet), all metadata associated with the array will be lost.

Examples

```
>>> arr = ndtest((2, 2, 2))
>>> arr
  a  b\c  c0  c1
a0  b0   0   1
a0  b1   2   3
a1  b0   4   5
a1  b1   6   7
>>> arr.to_frame()
  c      c0  c1
a  b
a0 b0   0   1
   b1   2   3
a1 b0   4   5
   b1   6   7
>>> arr.to_frame(fold_last_axis_name=True)
      c0  c1
a  b\c
a0 b0   0   1
   b1   2   3
a1 b0   4   5
   b1   6   7
```

Plotting

*Array.plot*Plots the data of the array into a graph (window pop-up).

`larray.Array.plot`

property `Array.plot`

Plots the data of the array into a graph (window pop-up).

The graph can be tweaked to achieve the desired formatting and can be saved to a .png file.

Parameters

kind [str]

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot

- ‘density’ : same as ‘kde’
- ‘area’ : area plot
- ‘pie’ : pie plot
- ‘scatter’ : scatter plot (if array’s dimensions ≥ 2)
- ‘hexbin’ : hexbin plot (if array’s dimensions ≥ 2)

ax [matplotlib axes object, default None]

subplots [boolean, default False] Make separate subplots for each column

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in; Be aware, that passing in both an ax and sharex=True will alter all x axis labels for all axis in a figure!

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible

layout [tuple (optional)] (rows, columns) for the layout of subplots

figsize [a tuple (width, height) in inches]

use_index [boolean, default True] Use index as ticks for x axis

title [string] Title to use for the plot

grid [boolean, default None (matlab style default)] Axis grid lines

legend [False/True/‘reverse’] Place legend on axis subplots

style [list or dict] matplotlib line style per column

logx [boolean, default False] Use log scaling on x axis

logy [boolean, default False] Use log scaling on y axis

loglog [boolean, default False] Use log scaling on both x and y axes

xticks [sequence] Values to use for the xticks

yticks [sequence] Values to use for the yticks

xlim [2-tuple/list]

ylim [2-tuple/list]

rot [int, default None] Rotation for ticks (xticks for vertical, yticks for horizontal plots)

fontsize [int, default None] Font size for xticks and yticks

colormap [str or matplotlib colormap object, default None] Colormap to select colors from. If string, load colormap with that name from matplotlib.

colorbar [boolean, optional] If True, plot colorbar (only relevant for ‘scatter’ and ‘hexbin’ plots)

position [float] Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

layout [tuple (optional)] (rows, columns) for the layout of the plot

yerr [array-like] Error bars on y axis

xerr [array-like] Error bars on x axis

stacked [boolean, default False in line and bar plots, and True in area plot.] If True, create stacked plot.

****kwargs** [keywords] Options to pass to matplotlib plotting method

Returns

axes [matplotlib.AxesSubplot or np.array of them]

Notes

See Pandas documentation of *plot* function for more details on this subject

Examples

```
>>> import matplotlib.pyplot as plt
>>> a = ndtest('sex=M,F;age=0..20')
```

Simple line plot

```
>>> a.plot()
>>> # shows figure (reset the current figure after showing it! Do not call it_
↳before savefig)
>>> plt.show()
```

Line plot with grid, title and both axes in logscale

```
>>> a.plot(grid=True, loglog=True, title='line plot')
>>> # saves figure in a file (see matplotlib.pyplot.savefig documentation for_
↳more details)
>>> plt.savefig('my_file.png')
```

2 bar plots sharing the same x axis (one for males and one for females)

```
>>> a.plot.bar(subplots=True, sharex=True)
>>> plt.show()
```

Create a figure containing 2 x 2 graphs

```
>>> # see matplotlib.pyplot.subplots documentation for more details
>>> fig, ax = plt.subplots(2, 2, figsize=(15, 15))
>>> # 2 curves : Males and Females
>>> a.plot(ax=ax[0, 0], title='line plot')
>>> # bar plot with stacked values
>>> a.plot.bar(ax=ax[0, 1], stacked=True, title='stacked bar plot')
>>> # same as previously but with colored areas instead of bars
>>> a.plot.area(ax=ax[1, 0], title='area plot')
>>> # scatter plot
>>> a.plot.scatter(ax=ax[1, 1], x='M', y='F', title='scatter plot')
>>> plt.show()
```

4.3.6 Utility Functions

- *Miscellaneous*
- *Rounding*
- *Exponents And Logarithms*
- *Trigonometric functions*
- *Hyperbolic functions*
- *Complex Numbers*
- *Floating Point Routines*

Miscellaneous

<code>where(condition, x, y)</code>	Return elements, either from <i>x</i> or <i>y</i> , depending on <i>condition</i> .
<code>maximum(x1, x2[, out, dtype])</code>	Element-wise maximum of array elements.
<code>minimum(x1, x2[, out, dtype])</code>	Element-wise minimum of array elements.
<code>inverse(*args, **kwargs)</code>	Compute the (multiplicative) inverse of a matrix.
<code>interp(*args, **kwargs)</code>	One-dimensional linear interpolation.
<code>convolve(*args, **kwargs)</code>	Returns the discrete, linear convolution of two one-dimensional sequences.
<code>absolute(x, /[, out, where, casting, order, ...])</code>	Calculate the absolute value element-wise.
<code>fabs(x, /[, out, where, casting, order, ...])</code>	Compute the absolute values element-wise.
<code>isnan(x, /[, out, where, casting, order, ...])</code>	Test element-wise for NaN and return result as a boolean array.
<code>isinf(x, /[, out, where, casting, order, ...])</code>	Test element-wise for positive or negative infinity.
<code>nan_to_num(*args, **kwargs)</code>	Replace NaN with zero and infinity with large finite numbers (default behaviour) or with the numbers defined by the user using the <i>nan</i> , <i>posinf</i> and/or <i>neginf</i> keywords.
<code>sqrt(x, /[, out, where, casting, order, ...])</code>	Return the non-negative square-root of an array, element-wise.
<code>i0(*args, **kwargs)</code>	Modified Bessel function of the first kind, order 0.
<code>sinc(*args, **kwargs)</code>	Return the sinc function.

larray.where

`larray.where(condition, x, y)`

Return elements, either from *x* or *y*, depending on *condition*.

Parameters

condition [boolean Array] When True, yield *x*, otherwise yield *y*.

x, y [Array] Values from which to choose.

Returns

out [Array] If both *x* and *y* are specified, the output array contains elements of *x* where *condition* is True, and elements from *y* elsewhere.

Examples

```
>>> from larray import Array
>>> arr = Array([[10, 7, 5, 9],
...              [5, 8, 3, 7],
...              [6, 2, 0, 9],
...              [9, 10, 5, 6]], "a=a0..a3;b=b0..b3")
>>> arr
a\b  b0  b1  b2  b3
a0   10   7   5   9
a1    5   8   3   7
a2    6   2   0   9
a3    9  10   5   6
```

Simple use

```
>>> where(arr <= 5, 0, arr)
a\b  b0  b1  b2  b3
a0   10   7   0   9
a1    0   8   0   7
a2    6   0   0   9
a3    9  10   0   6
```

With broadcasting

```
>>> mean_by_col = arr.mean('a')
>>> mean_by_col
b  b0    b1    b2    b3
   7.5  6.75  3.25  7.75
>>> # for each column, set values below the mean value to the mean value
>>> where(arr < mean_by_col, mean_by_col, arr)
a\b  b0    b1    b2    b3
a0  10.0   7.0   5.0   9.0
a1   7.5   8.0   3.25  7.75
a2   7.5   6.75  3.25   9.0
a3   9.0  10.0   5.0   7.75
```

larray.maximum

`larray.maximum(x1, x2, out=None, dtype=None)`

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

Parameters

x1, x2 [Array] The arrays holding the elements to be compared.

out [Array, optional] An array into which the result is stored.

dtype [data-type, optional] Overrides the dtype of the output array.

Returns

y [Array or scalar] The maximum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

See also:

minimum Element-wise minimum of two arrays, propagates NaNs.

Notes

The maximum is equivalent to `where(x1 >= x2, x1, x2)` when neither `x1` nor `x2` are NaNs, but it is faster.

Examples

```
>>> from larray import Array
>>> arr1 = Array([[10, 7, 5, 9],
...               [5, 8, 3, 7]], "a=a0,a1;b=b0..b3")
>>> arr2 = Array([[6, 2, 9, 0],
...               [9, 10, 5, 6]], "a=a0,a1;b=b0..b3")
>>> arr1
a\b  b0  b1  b2  b3
a0   10   7   5   9
a1    5   8   3   7
>>> arr2
a\b  b0  b1  b2  b3
a0    6   2   9   0
a1    9  10   5   6
```

```
>>> maximum(arr1, arr2)
a\b  b0  b1  b2  b3
a0   10   7   9   9
a1    9  10   5   7
```

With broadcasting

```
>>> arr2['a0']
b  b0  b1  b2  b3
   6   2   9   0
>>> maximum(arr1, arr2['a0'])
a\b  b0  b1  b2  b3
a0   10   7   9   9
a1    6   8   9   7
```

larray.minimum

`larray.minimum(x1, x2, out=None, dtype=None)`

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

Parameters

x1, x2 [Array] The arrays holding the elements to be compared.

out [Array, optional] An array into which the result is stored.

dtype [data-type, optional] Overrides the dtype of the output array.

Returns

y [Array or scalar] The minimum of $x1$ and $x2$, element-wise. This is a scalar if both $x1$ and $x2$ are scalars.

See also:

[*maximum*](#) Element-wise maximum of two arrays, propagates NaNs.

Notes

The minimum is equivalent to `where(x1 <= x2, x1, x2)` when neither $x1$ nor $x2$ are NaNs, but it is faster.

Examples

```
>>> from larray import Array
>>> arr1 = Array([[10, 7, 5, 9],
...               [5, 8, 3, 7]], "a=a0,a1;b=b0..b3")
>>> arr2 = Array([[6, 2, 9, 0],
...               [9, 10, 5, 6]], "a=a0,a1;b=b0..b3")
>>> arr1
a\b  b0  b1  b2  b3
a0   10   7   5   9
a1    5   8   3   7
>>> arr2
a\b  b0  b1  b2  b3
a0    6   2   9   0
a1    9  10   5   6
```

```
>>> minimum(arr1, arr2)
a\b  b0  b1  b2  b3
a0    6   2   5   0
a1    5   8   3   6
```

With broadcasting

```
>>> arr2['a0']
b  b0  b1  b2  b3
   6   2   9   0
>>> minimum(arr1, arr2['a0'])
a\b  b0  b1  b2  b3
a0    6   2   5   0
a1    5   2   3   0
```

larray.inverse

`larray.inverse(*args, **kwargs)`

Compute the (multiplicative) inverse of a matrix.

larray specific variant of `numpy.inv`.

Documentation from `numpy`:

Given a square matrix *a*, return the matrix *ainv* satisfying `dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])`.

Parameters

a [(..., M, M) array_like] Matrix to be inverted.

Returns

ainv [(..., M, M) ndarray or matrix] (Multiplicative) inverse of the matrix *a*.

Raises

LinAlgError If *a* is not square or inversion fails.

Notes

New in version 1.8.0.

Broadcasting rules apply, see the `numpy.linalg` documentation for details.

Examples

```
>>> from numpy.linalg import inv
>>> a = np.array([[1., 2.], [3., 4.]])
>>> ainv = inv(a)
>>> np.allclose(np.dot(a, ainv), np.eye(2))
True
>>> np.allclose(np.dot(ainv, a), np.eye(2))
True
```

If *a* is a matrix object, then the return value is a matrix as well:

```
>>> ainv = inv(np.matrix(a))
>>> ainv
matrix([[ -2. ,  1. ],
        [ 1.5, -0.5]])
```

Inverses of several matrices can be computed at once:

```
>>> a = np.array([[[1., 2.], [3., 4.]], [[1, 3], [3, 5]]])
>>> inv(a)
array([[[ -2. ,  1. ],
        [ 1.5 , -0.5 ]],
       [[-1.25,  0.75],
        [ 0.75, -0.25]]])
```

larray.interp

`larray.interp(*args, **kwargs)`

One-dimensional linear interpolation.

larray specific variant of `numpy.interp`.

Documentation from `numpy`:

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (xp, fp) , evaluated at x .

Parameters

x [array_like] The x-coordinates at which to evaluate the interpolated values.

xp [1-D sequence of floats] The x-coordinates of the data points, must be increasing if argument *period* is not specified. Otherwise, *xp* is internally sorted after normalizing the periodic boundaries with `xp = xp % period`.

fp [1-D sequence of float or complex] The y-coordinates of the data points, same length as *xp*.

left [optional float or complex corresponding to fp] Value to return for $x < xp[0]$, default is *fp[0]*.

right [optional float or complex corresponding to fp] Value to return for $x > xp[-1]$, default is *fp[-1]*.

period [None or float, optional] A period for the x-coordinates. This parameter allows the proper interpolation of angular x-coordinates. Parameters *left* and *right* are ignored if *period* is specified.

New in version 1.10.0.

Returns

y [float or complex (corresponding to fp) or ndarray] The interpolated values, same shape as *x*.

Raises

ValueError If *xp* and *fp* have different length If *xp* or *fp* are not 1-D sequences If *period* == 0

Notes

The x-coordinate sequence is expected to be increasing, but this is not explicitly enforced. However, if the sequence *xp* is non-increasing, interpolation results are meaningless.

Note that, since NaN is unsortable, *xp* also cannot contain NaNs.

A simple check for *xp* being strictly increasing is:

```
np.all(np.diff(xp) > 0)
```

Examples

```
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
array([3. , 3. , 2.5 , 0.56, 0. ])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0
```

Plot an interpolant to the sine function:

```
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> xvals = np.linspace(0, 2*np.pi, 50)
>>> yinterp = np.interp(xvals, x, y)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(xvals, yinterp, '-x')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.show()
```

Interpolation with periodic x-coordinates:

```
>>> x = [-180, -170, -185, 185, -10, -5, 0, 365]
>>> xp = [190, -190, 350, -350]
>>> fp = [5, 10, 3, 4]
>>> np.interp(x, xp, fp, period=360)
array([7.5 , 5. , 8.75, 6.25, 3. , 3.25, 3.5 , 3.75])
```

Complex interpolation:

```
>>> x = [1.5, 4.0]
>>> xp = [2, 3, 5]
>>> fp = [1.0j, 0, 2+3j]
>>> np.interp(x, xp, fp)
array([0.+1.j , 1.+1.5j])
```

larray.convolve

`larray.convolve(*args, **kwargs)`

Returns the discrete, linear convolution of two one-dimensional sequences.

`larray` specific variant of `numpy.convolve`.

Documentation from `numpy`:

The convolution operator is often seen in signal processing, where it models the effect of a linear time-invariant system on a signal [1]. In probability theory, the sum of two independent random variables is distributed according to the convolution of their individual distributions.

If `v` is longer than `a`, the arrays are swapped before computation.

Parameters

a [(N,) array_like] First one-dimensional input array.

v [(M,) array_like] Second one-dimensional input array.

mode [{ 'full', 'valid', 'same' }, optional]

‘full’: By default, mode is ‘full’. This returns the convolution at each point of overlap, with an output shape of $(N+M-1,)$. At the end-points of the convolution, the signals do not overlap completely, and boundary effects may be seen.

‘same’: Mode ‘same’ returns output of length $\max(M, N)$. Boundary effects are still visible.

‘valid’: Mode ‘valid’ returns output of length $\max(M, N) - \min(M, N) + 1$. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.

Returns

out [ndarray] Discrete, linear convolution of *a* and *v*.

See also:

scipy.signal.fftconvolve Convolve two arrays using the Fast Fourier Transform.

scipy.linalg.toeplitz Used to construct the convolution operator.

polymul Polynomial multiplication. Same output as convolve, but also accepts poly1d objects as input.

Notes

The discrete convolution operation is defined as

$$(a * v)[n] = \sum_{m=-\infty}^{\infty} a[m]v[n - m]$$

It can be shown that a convolution $x(t) * y(t)$ in time/space is equivalent to the multiplication $X(f)Y(f)$ in the Fourier domain, after appropriate padding (padding is necessary to prevent circular convolution). Since multiplication is more efficient (faster) than convolution, the function *scipy.signal.fftconvolve* exploits the FFT to calculate the convolution of large data-sets.

References

[1]

Examples

Note how the convolution operator flips the second array before “sliding” the two across one another:

```
>>> np.convolve([1, 2, 3], [0, 1, 0.5])
array([0. , 1. , 2.5, 4. , 1.5])
```

Only return the middle values of the convolution. Contains boundary effects, where zeros are taken into account:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'same')
array([1. , 2.5, 4. ])
```

The two arrays are of the same length, so there is only one position where they completely overlap:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'valid')
array([2.5])
```

larray.absolute

`larray.absolute(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Calculate the absolute value element-wise.

larray specific variant of `numpy.absolute`.

Documentation from `numpy`:

`np.abs` is a shorthand for this function.

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

absolute [ndarray] An ndarray containing the absolute value of each element in *x*. For complex input, $a + ib$, the absolute value is $\sqrt{a^2 + b^2}$. This is a scalar if *x* is a scalar.

Examples

```
>>> x = np.array([-1.2, 1.2])
>>> np.absolute(x)
array([ 1.2,  1.2])
>>> np.absolute(1.2 + 1j)
1.5620499351813308
```

Plot the function over `[-10, 10]`:

```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(start=-10, stop=10, num=101)
>>> plt.plot(x, np.absolute(x))
>>> plt.show()
```

Plot the function over the complex plane:

```
>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10], cmap='gray')
>>> plt.show()
```

larray.fabs

`larray.fabs(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Compute the absolute values element-wise.

larray specific variant of `numpy.fabs`.

Documentation from `numpy`:

This function returns the absolute values (positive magnitude) of the data in *x*. Complex values are not handled, use *absolute* to find the absolute values of complex data.

Parameters

x [array_like] The array of numbers for which the absolute values are required. If *x* is a scalar, the result *y* will also be a scalar.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The absolute values of *x*, the returned values are always floats. This is a scalar if *x* is a scalar.

See also:

[*absolute*](#) Absolute values including *complex* types.

Examples

```
>>> np.fabs(-1)
1.0
>>> np.fabs([-1.2, 1.2])
array([ 1.2,  1.2])
```


larray.isnan

`larray.isnan(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Test element-wise for NaN and return result as a boolean array.

larray specific variant of `numpy.isnan`.

Documentation from `numpy`:

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or bool] True where *x* is NaN, false otherwise. This is a scalar if *x* is a scalar.

See also:

[`isinf`](#), [`isneginf`](#), [`isposinf`](#), [`isfinite`](#), [`isnat`](#)

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

```
>>> np.isnan(np.nan)
True
>>> np.isnan(np.inf)
False
>>> np.isnan([np.log(-1.), 1., np.log(0)])
array([ True, False, False])
```

larray.isinf

`larray.isinf(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Test element-wise for positive or negative infinity.

larray specific variant of `numpy.isinf`.

Documentation from `numpy`:

Returns a boolean array of the same shape as *x*, True where $x == +/-inf$, otherwise False.

Parameters

x [array_like] Input values

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [bool (scalar) or boolean ndarray] True where *x* is positive or negative infinity, false otherwise. This is a scalar if *x* is a scalar.

See also:

`isneginf`, `isposinf`, `isnan`, `isfinite`

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is supplied when the first argument is a scalar, or if the first and second arguments have different shapes.

Examples

```
>>> np.isinf(np.inf)
True
>>> np.isinf(np.nan)
False
>>> np.isinf(np.NINF)
True
>>> np.isinf([np.inf, -np.inf, 1.0, np.nan])
array([ True,  True, False, False])
```

```

>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isinf(x, y)
array([1, 0, 1])
>>> y
array([1, 0, 1])

```

larray.nan_to_num

`larray.nan_to_num(*args, **kwargs)`

Replace NaN with zero and infinity with large finite numbers (default behaviour) or with the numbers defined by the user using the *nan*, *posinf* and/or *neginf* keywords.

larray specific variant of `numpy.nan_to_num`.

Documentation from `numpy`:

If *x* is inexact, NaN is replaced by zero or by the user defined value in *nan* keyword, infinity is replaced by the largest finite floating point values representable by *x.dtype* or by the user defined value in *posinf* keyword and -infinity is replaced by the most negative finite floating point values representable by *x.dtype* or by the user defined value in *neginf* keyword.

For complex dtypes, the above is applied to each of the real and imaginary components of *x* separately.

If *x* is not inexact, then no replacements are made.

Parameters

x [scalar or array_like] Input data.

copy [bool, optional] Whether to create a copy of *x* (True) or to replace values in-place (False). The in-place operation only occurs if casting to an array does not require a copy. Default is True.

New in version 1.13.

nan [int, float, optional] Value to be used to fill NaN values. If no value is passed then NaN values will be replaced with 0.0.

New in version 1.17.

posinf [int, float, optional] Value to be used to fill positive infinity values. If no value is passed then positive infinity values will be replaced with a very large number.

New in version 1.17.

neginf [int, float, optional] Value to be used to fill negative infinity values. If no value is passed then negative infinity values will be replaced with a very small (or negative) number.

New in version 1.17.

Returns

out [ndarray] *x*, with the non-finite values replaced. If *copy* is False, this may be *x* itself.

See also:

isinf Shows which elements are positive or negative infinity.

isneginf Shows which elements are negative infinity.

isposinf Shows which elements are positive infinity.

isnan Shows which elements are Not a Number (NaN).

isfinite Shows which elements are finite (not NaN, not infinity)

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

```
>>> np.nan_to_num(np.inf)
1.7976931348623157e+308
>>> np.nan_to_num(-np.inf)
-1.7976931348623157e+308
>>> np.nan_to_num(np.nan)
0.0
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
>>> np.nan_to_num(x)
array([ 1.79769313e+308, -1.79769313e+308,  0.00000000e+000, # may vary
        -1.28000000e+002,  1.28000000e+002])
>>> np.nan_to_num(x, nan=-9999, posinf=33333333, neginf=33333333)
array([ 3.3333333e+07,  3.3333333e+07, -9.9990000e+03,
        -1.2800000e+02,  1.2800000e+02])
>>> y = np.array([complex(np.inf, np.nan), np.nan, complex(np.nan, np.inf)])
array([ 1.79769313e+308, -1.79769313e+308,  0.00000000e+000, # may vary
        -1.28000000e+002,  1.28000000e+002])
>>> np.nan_to_num(y)
array([ 1.79769313e+308 +0.00000000e+000j, # may vary
        0.00000000e+000 +0.00000000e+000j,
        0.00000000e+000 +1.79769313e+308j])
>>> np.nan_to_num(y, nan=111111, posinf=222222)
array([222222.+111111.j, 111111.      +0.j, 111111.+222222.j])
```

larray.sqrt

`larray.sqrt(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Return the non-negative square-root of an array, element-wise.

larray specific variant of `numpy.sqrt`.

Documentation from `numpy`:

Parameters

x [array_like] The values whose square-roots are required.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out*=None, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] An array of the same shape as *x*, containing the positive square-root of each element in *x*. If any element in *x* is complex, a complex array is returned (and the square-roots of negative reals are calculated). If all of the elements in *x* are real, so is *y*, with negative elements returning nan. If *out* was provided, *y* is a reference to it. This is a scalar if *x* is a scalar.

See also:

lib.scimath.sqrt A version which returns complex numbers when given negative reals.

Notes

sqrt has—consistent with common convention—as its branch cut the real “interval” $[-inf, 0)$, and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

Examples

```
>>> np.sqrt([1, 4, 9])
array([ 1.,  2.,  3.]
```

```
>>> np.sqrt([4, -1, -3+4j])
array([ 2.+0.j,  0.+1.j,  1.+2.j])
```

```
>>> np.sqrt([4, -1, np.inf])
array([ 2., nan, inf])
```

larray.i0

`larray.i0(*args, **kwargs)`

Modified Bessel function of the first kind, order 0.

`larray` specific variant of `numpy.i0`.

Documentation from `numpy`:

Usually denoted I_0 . This function does broadcast, but will *not* “up-cast” int dtype arguments unless accompanied by at least one float or complex dtype argument (see Raises below).

Parameters

x [array_like, dtype float or complex] Argument of the Bessel function.

Returns

out [ndarray, shape = x.shape, dtype = x.dtype] The modified Bessel function evaluated at each of the elements of *x*.

Raises

TypeError: array cannot be safely cast to required type If argument consists exclusively of int dtypes.

See also:

`scipy.special.i0`, `scipy.special.iv`, `scipy.special.ive`

Notes

The scipy implementation is recommended over this function: it is a proper ufunc written in C, and more than an order of magnitude faster.

We use the algorithm published by Clenshaw [1] and referenced by Abramowitz and Stegun [2], for which the function domain is partitioned into the two intervals [0,8] and (8,inf), and Chebyshev polynomial expansions are employed in each interval. Relative error on the domain [0,30] using IEEE arithmetic is documented [3] as having a peak of 5.8e-16 with an rms of 1.4e-16 (n = 30000).

References

[1], [2], [3]

Examples

```
>>> np.i0(0.)
array(1.0) # may vary
>>> np.i0([0., 1. + 2j])
array([ 1.00000000+0.j          ,  0.18785373+0.64616944j]) # may vary
```

larray.sinc

`larray.sinc(*args, **kwargs)`

Return the sinc function.

larray specific variant of `numpy.sinc`.

Documentation from numpy:

The sinc function is $\sin(\pi x)/(\pi x)$.

Parameters

x [ndarray] Array (possibly multi-dimensional) of values for which to calculate `sinc(x)`.

Returns

out [ndarray] `sinc(x)`, which has the same shape as the input.

Notes

`sinc(0)` is the limit value 1.

The name `sinc` is short for “sine cardinal” or “sinus cardinalis”.

The `sinc` function is used in various signal processing applications, including in anti-aliasing, in the construction of a Lanczos resampling filter, and in interpolation.

For bandlimited interpolation of discrete-time signals, the ideal interpolation kernel is proportional to the `sinc` function.

References

[1], [2]

Examples

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 41)
>>> np.sinc(x)
array([-3.89804309e-17, -4.92362781e-02, -8.40918587e-02, # may vary
       -8.90384387e-02, -5.84680802e-02,  3.89804309e-17,
        6.68206631e-02,  1.16434881e-01,  1.26137788e-01,
        8.50444803e-02, -3.89804309e-17, -1.03943254e-01,
       -1.89206682e-01, -2.16236208e-01, -1.55914881e-01,
        3.89804309e-17,  2.33872321e-01,  5.04551152e-01,
        7.56826729e-01,  9.35489284e-01,  1.00000000e+00,
        9.35489284e-01,  7.56826729e-01,  5.04551152e-01,
        2.33872321e-01,  3.89804309e-17, -1.55914881e-01,
       -2.16236208e-01, -1.89206682e-01, -1.03943254e-01,
       -3.89804309e-17,  8.50444803e-02,  1.26137788e-01,
        1.16434881e-01,  6.68206631e-02,  3.89804309e-17,
       -5.84680802e-02, -8.90384387e-02, -8.40918587e-02,
       -4.92362781e-02, -3.89804309e-17])
```

```
>>> plt.plot(x, np.sinc(x))
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Sinc Function")
Text(0.5, 1.0, 'Sinc Function')
>>> plt.ylabel("Amplitude")
Text(0, 0.5, 'Amplitude')
>>> plt.xlabel("X")
Text(0.5, 0, 'X')
>>> plt.show()
```

Rounding

<code>round(*args, **kwargs)</code>	Round an array to the given number of decimals.
<code>floor(x, /[, out, where, casting, order, ...])</code>	Return the floor of the input, element-wise.
<code>ceil(x, /[, out, where, casting, order, ...])</code>	Return the ceiling of the input, element-wise.
<code>trunc(x, /[, out, where, casting, order, ...])</code>	Return the truncated value of the input, element-wise.
<code>rint(x, /[, out, where, casting, order, ...])</code>	Round elements of the array to the nearest integer.
<code>fix(*args, **kwargs)</code>	Round to nearest integer towards zero.

`larray.round`

`larray.round(*args, **kwargs)`
Round an array to the given number of decimals.

larray specific variant of `numpy.round_`.

Documentation from numpy:

See also:

around equivalent function; see for details.

`larray.floor`

`larray.floor(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Return the floor of the input, element-wise.

larray specific variant of `numpy.floor`.

Documentation from numpy:

The floor of the scalar x is the largest integer i , such that $i \leq x$. It is often denoted as $\lfloor x \rfloor$.

Parameters

x [array_like] Input data.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The floor of each element in *x*. This is a scalar if *x* is a scalar.

See also:

`ceil`, `trunc`, `rint`

Notes

Some spreadsheet programs calculate the “floor-towards-zero”, in other words `floor(-2.5) == -2`. NumPy instead uses the definition of *floor* where `floor(-2.5) == -3`.

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.floor(a)
array([-2., -2., -1.,  0.,  1.,  1.,  2.])
```

larray.ceil

`larray.ceil(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Return the ceiling of the input, element-wise.

larray specific variant of `numpy.ceil`.

Documentation from `numpy`:

The ceil of the scalar x is the smallest integer i , such that $i \geq x$. It is often denoted as $\lceil x \rceil$.

Parameters

x [array_like] Input data.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The ceiling of each element in *x*, with *float* dtype. This is a scalar if *x* is a scalar.

See also:

floor, *trunc*, *rint*

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.ceil(a)
array([-1., -1., -0., 1., 2., 2., 2.])
```

larray.trunc

`larray.trunc(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Return the truncated value of the input, element-wise.

larray specific variant of `numpy.trunc`.

Documentation from `numpy`:

The truncated value of the scalar x is the nearest integer i which is closer to zero than x is. In short, the fractional part of the signed number x is discarded.

Parameters

x [array_like] Input data.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The truncated value of each element in x . This is a scalar if x is a scalar.

See also:

[*ceil*](#), [*floor*](#), [*rint*](#)

Notes

New in version 1.3.0.

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.trunc(a)
array([-1., -1., -0., 0., 1., 1., 2.])
```

larray.rint

`larray.rint(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Round elements of the array to the nearest integer.

larray specific variant of `numpy rint`.

Documentation from `numpy`:

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Output array is same shape and type as *x*. This is a scalar if *x* is a scalar.

See also:

[*ceil*](#), [*floor*](#), [*trunc*](#)

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.rint(a)
array([-2., -2., -0., 0., 2., 2., 2.])
```

larray.fix

`larray.fix(*args, **kwargs)`

Round to nearest integer towards zero.

`larray` specific variant of `numpy.fix`.

Documentation from `numpy`:

Round an array of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

Parameters

x [array_like] An array of floats to be rounded

y [ndarray, optional] Output array

Returns

out [ndarray of floats] The array of rounded numbers

See also:

[`trunc`](#), [`floor`](#), [`ceil`](#)

around Round to given number of decimals

Examples

```
>>> np.fix(3.14)
3.0
>>> np.fix(3)
3.0
>>> np.fix([2.1, 2.9, -2.1, -2.9])
array([ 2.,  2., -2., -2.]
```

Exponents And Logarithms

<code>exp(x, /[, out, where, casting, order, ...])</code>	Calculate the exponential of all elements in the input array.
<code>expm1(x, /[, out, where, casting, order, ...])</code>	Calculate $\exp(x) - 1$ for all elements in the array.
<code>exp2(x, /[, out, where, casting, order, ...])</code>	Calculate 2^{**p} for all p in the input array.
<code>log(x, /[, out, where, casting, order, ...])</code>	Natural logarithm, element-wise.
<code>log10(x, /[, out, where, casting, order, ...])</code>	Return the base 10 logarithm of the input array, element-wise.
<code>log2(x, /[, out, where, casting, order, ...])</code>	Base-2 logarithm of x .
<code>log1p(x, /[, out, where, casting, order, ...])</code>	Return the natural logarithm of one plus the input array, element-wise.
<code>logaddexp(x1, x2, /[, out, where, casting, ...])</code>	Logarithm of the sum of exponentiations of the inputs.
<code>logaddexp2(x1, x2, /[, out, where, casting, ...])</code>	Logarithm of the sum of exponentiations of the inputs in base-2.

larray.exp

```
larray.exp(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[,
           signature, extobj])
```

Calculate the exponential of all elements in the input array.

larray specific variant of `numpy.exp`.

Documentation from `numpy`:

Parameters

x [array_like] Input values.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Output array, element-wise exponential of *x*. This is a scalar if *x* is a scalar.

See also:

expm1 Calculate $\exp(x) - 1$ for all elements in the array.

exp2 Calculate 2^{**x} for all elements in the array.

Notes

The irrational number e is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, $\exp(x)$ is always positive.

For complex arguments, $x = a + ib$, we can write $e^x = e^a e^{ib}$. The first term, e^a , is already known (it is the real argument, described above). The second term, e^{ib} , is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

References

[1], [2]

Examples

Plot the magnitude and phase of $\exp(x)$ in the complex plane:

```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)
>>> xx = x + 1j * x[:, np.newaxis] # a + ib over complex plane
>>> out = np.exp(xx)
```

```
>>> plt.subplot(121)
>>> plt.imshow(np.abs(out),
...           extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='gray')
>>> plt.title('Magnitude of exp(x)')
```

```
>>> plt.subplot(122)
>>> plt.imshow(np.angle(out),
...           extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='hsv')
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```

larray.expm1

`larray.expm1(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Calculate $\exp(x) - 1$ for all elements in the array.

larray specific variant of `numpy.expm1`.

Documentation from `numpy`:

Parameters

x [array_like] Input values.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Element-wise exponential minus one: $\text{out} = \exp(x) - 1$. This is a scalar if *x* is a scalar.

See also:

[*log1p*](#) $\log(1 + x)$, the inverse of `expm1`.

Notes

This function provides greater precision than $\exp(x) - 1$ for small values of x .

Examples

The true value of $\exp(1e-10) - 1$ is $1.000000000005e-10$ to about 32 significant digits. This example shows the superiority of `expm1` in this case.

```
>>> np.expm1(1e-10)
1.000000000005e-10
>>> np.exp(1e-10) - 1
1.0000000082740371e-10
```

larray.exp2

`larray.exp2(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Calculate 2^{**p} for all p in the input array.

`larray` specific variant of `numpy.exp2`.

Documentation from `numpy`:

Parameters

x [array_like] Input values.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Element-wise 2 to the power x . This is a scalar if x is a scalar.

See also:

`power`

Notes

New in version 1.3.0.

Examples

```
>>> np.exp2([2, 3])
array([ 4.,  8.])
```

larray.log

`larray.log(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Natural logarithm, element-wise.

`larray` specific variant of `numpy.log`.

Documentation from `numpy`:

The natural logarithm *log* is the inverse of the exponential function, so that $\log(\exp(x)) = x$. The natural logarithm is logarithm in base *e*.

Parameters

x [array_like] Input value.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The natural logarithm of *x*, element-wise. This is a scalar if *x* is a scalar.

See also:

[log10](#), [log2](#), [log1p](#), [emath.log](#)

Notes

Logarithm is a multivalued function: for each x there is an infinite number of z such that $\exp(z) = x$. The convention is to return the z whose imaginary part lies in $[-\pi, \pi]$.

For real-valued input data types, *log* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log* is a complex analytical function that has a branch cut $[-\infty, 0]$ and is continuous from above on it. *log* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[1], [2]

Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([ 0.,  1.,  2., -Inf])
```

larray.log10

`larray.log10(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Return the base 10 logarithm of the input array, element-wise.

larray specific variant of `numpy.log10`.

Documentation from `numpy`:

Parameters

x [array_like] Input values.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The logarithm to the base 10 of *x*, element-wise. NaNs are returned where *x* is negative. This is a scalar if *x* is a scalar.

See also:

`emath.log10`

Notes

Logarithm is a multivalued function: for each x there is an infinite number of z such that $10^{**z} = x$. The convention is to return the z whose imaginary part lies in $[-\pi, \pi]$.

For real-valued input data types, `log10` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `log10` is a complex analytical function that has a branch cut $[-\infty, 0]$ and is continuous from above on it. `log10` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[1], [2]

Examples

```
>>> np.log10([1e-15, -3.])
array([-15.,  nan])
```

larray.log2

`larray.log2(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Base-2 logarithm of x .

`larray` specific variant of `numpy.log2`.

Documentation from `numpy`:

Parameters

x [array_like] Input values.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is `True`, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] Base-2 logarithm of x . This is a scalar if x is a scalar.

See also:

[log](#), [log10](#), [log1p](#), [emath.log2](#)

Notes

New in version 1.3.0.

Logarithm is a multivalued function: for each x there is an infinite number of z such that $2^{**z} = x$. The convention is to return the z whose imaginary part lies in $[-\pi, \pi]$.

For real-valued input data types, `log2` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `log2` is a complex analytical function that has a branch cut $[-\infty, 0]$ and is continuous from above on it. `log2` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

Examples

```
>>> x = np.array([0, 1, 2, 2**4])
>>> np.log2(x)
array([-Inf,  0.,  1.,  4.])
```

```
>>> xi = np.array([0+1.j, 1, 2+0.j, 4.j])
>>> np.log2(xi)
array([ 0.+2.26618007j,  0.+0.j           ,  1.+0.j           ,  2.+2.26618007j])
```

larray.log1p

`larray.log1p(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Return the natural logarithm of one plus the input array, element-wise.

larray specific variant of `numpy.log1p`.

Documentation from `numpy`:

Calculates $\log(1 + x)$.

Parameters

x [array_like] Input values.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] Natural logarithm of $1 + x$, element-wise. This is a scalar if x is a scalar.

See also:

expm1 $\exp(x) - 1$, the inverse of *log1p*.

Notes

For real-valued input, *log1p* is accurate also for x so small that $1 + x == 1$ in floating-point accuracy.

Logarithm is a multivalued function: for each x there is an infinite number of z such that $\exp(z) = 1 + x$. The convention is to return the z whose imaginary part lies in $[-\pi, \pi]$.

For real-valued input data types, *log1p* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log1p* is a complex analytical function that has a branch cut $[-\infty, -1]$ and is continuous from above on it. *log1p* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[1], [2]

Examples

```
>>> np.log1p(1e-99)
1e-99
>>> np.log(1 + 1e-99)
0.0
```

larray.logaddexp

`larray.logaddexp(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Logarithm of the sum of exponentiations of the inputs.

larray specific variant of `numpy.logaddexp`.

Documentation from `numpy`:

Calculates $\log(\exp(x1) + \exp(x2))$. This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

Parameters

x1, x2 [array_like] Input values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the

default `out=None`, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

result [ndarray] Logarithm of $\exp(x1) + \exp(x2)$. This is a scalar if both `x1` and `x2` are scalars.

See also:

[`logaddexp2`](#) Logarithm of the sum of exponentiations of inputs in base 2.

Notes

New in version 1.3.0.

Examples

```
>>> prob1 = np.log(1e-50)
>>> prob2 = np.log(2.5e-50)
>>> prob12 = np.logaddexp(prob1, prob2)
>>> prob12
-113.87649168120691
>>> np.exp(prob12)
3.50000000000000057e-50
```

larray.logaddexp2

`larray.logaddexp2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Logarithm of the sum of exponentiations of the inputs in base-2.

larray specific variant of `numpy.logaddexp2`.

Documentation from `numpy`:

Calculates $\log_2(2^{x1} + 2^{x2})$. This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

Parameters

x1, x2 [array_like] Input values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is `True`, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the

default `out=None`, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

result [ndarray] Base-2 logarithm of $2^{x1} + 2^{x2}$. This is a scalar if both *x1* and *x2* are scalars.

See also:

[*logaddexp*](#) Logarithm of the sum of exponentiations of the inputs.

Notes

New in version 1.3.0.

Examples

```
>>> prob1 = np.log2(1e-50)
>>> prob2 = np.log2(2.5e-50)
>>> prob12 = np.logaddexp2(prob1, prob2)
>>> prob1, prob2, prob12
(-166.09640474436813, -164.77447664948076, -164.28904982231052)
>>> 2**prob12
3.4999999999999914e-50
```

Trigonometric functions

<code>sin(x, /[, out, where, casting, order, ...])</code>	Trigonometric sine, element-wise.
<code>cos(x, /[, out, where, casting, order, ...])</code>	Cosine element-wise.
<code>tan(x, /[, out, where, casting, order, ...])</code>	Compute tangent element-wise.
<code>arcsin(x, /[, out, where, casting, order, ...])</code>	Inverse sine, element-wise.
<code>arccos(x, /[, out, where, casting, order, ...])</code>	Trigonometric inverse cosine, element-wise.
<code>arctan(x, /[, out, where, casting, order, ...])</code>	Trigonometric inverse tangent, element-wise.
<code>hypot(x1, x2, /[, out, where, casting, ...])</code>	Given the “legs” of a right triangle, return its hypotenuse.
<code>arctan2(x1, x2, /[, out, where, casting, ...])</code>	Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.
<code>degrees(x, /[, out, where, casting, order, ...])</code>	Convert angles from radians to degrees.
<code>radians(x, /[, out, where, casting, order, ...])</code>	Convert angles from degrees to radians.
<code>unwrap(*args, **kwargs)</code>	Unwrap by changing deltas between values to 2π complement.

larray.sin

`larray.sin(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Trigonometric sine, element-wise.

larray specific variant of `numpy.sin`.

Documentation from `numpy`:

Parameters

x [array_like] Angle, in radians (2π rad equals 360 degrees).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [array_like] The sine of each element of *x*. This is a scalar if *x* is a scalar.

See also:

[*arcsin*](#), [*sinh*](#), [*cos*](#)

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π . Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Examples

Print sine of one angle:

```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.          ,  0.5          ,  0.70710678,  0.8660254 ,  1.          ])
```

Plot the sine function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

larray.cos

`larray.cos(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`
Cosine element-wise.

larray specific variant of `numpy.cos`.

Documentation from `numpy`:

Parameters

x [array_like] Input array in radians.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The corresponding cosine values. This is a scalar if *x* is a scalar.

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([ 1.00000000e+00,  6.12303177e-17, -1.00000000e+00])
>>>
>>> # Example of providing the optional output parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)), np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

larray.tan

`larray.tan(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Compute tangent element-wise.

larray specific variant of `numpy.tan`.

Documentation from `numpy`:

Equivalent to `np.sin(x)/np.cos(x)` element-wise.

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The corresponding tangent values. This is a scalar if *x* is a scalar.

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> from math import pi
>>> np.tan(np.array([-pi,pi/2,pi]))
array([ 1.22460635e-16,  1.63317787e+16, -1.22460635e-16])
>>>
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

larray.arcsin

`larray.arcsin(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Inverse sine, element-wise.

larray specific variant of `numpy.arcsin`.

Documentation from `numpy`:

Parameters

x [array_like] y-coordinate on the unit circle.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

angle [ndarray] The inverse sine of each element in x , in radians and in the closed interval $[-\pi/2, \pi/2]$. This is a scalar if x is a scalar.

See also:

sin, *cos*, *arccos*, *tan*, *arctan*, *arctan2*, `math.arcsin`

Notes

arcsin is a multivalued function: for each x there are infinitely many numbers z such that $\sin(z) = x$. The convention is to return the angle z whose real part lies in $[-\pi/2, \pi/2]$.

For real-valued input data types, *arcsin* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arcsin* is a complex analytic function that has, by convention, the branch cuts $[-\infty, -1]$ and $[1, \infty]$ and is continuous from above on the former and from below on the latter.

The inverse sine is also known as *asin* or \sin^{-1} .

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. <http://www.math.sfu.ca/~cbm/aands/>

Examples

```
>>> np.arcsin(1)      # pi/2
1.5707963267948966
>>> np.arcsin(-1)     # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
```

larray.arccos

`larray.arccos(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Trigonometric inverse cosine, element-wise.

larray specific variant of `numpy.arccos`.

Documentation from `numpy`:

The inverse of *cos* so that, if $y = \cos(x)$, then $x = \arccos(y)$.

Parameters

x [array_like] x -coordinate on the unit circle. For real arguments, the domain is $[-1, 1]$.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

angle [ndarray] The angle of the ray intersecting the unit circle at the given *x*-coordinate in radians $[0, \pi]$. This is a scalar if *x* is a scalar.

See also:

cos, *arctan*, *arcsin*, *emath.arccos*

Notes

arccos is a multivalued function: for each *x* there are infinitely many numbers *z* such that $\cos(z) = x$. The convention is to return the angle *z* whose real part lies in $[0, \pi]$.

For real-valued input data types, *arccos* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields *nan* and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytic function that has branch cuts $[-inf, -1]$ and $[1, inf]$ and is continuous from above on the former and from below on the latter.

The inverse *cos* is also known as *acos* or \cos^{-1} .

References

M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 79. <http://www.math.sfu.ca/~cbm/aands/>

Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> np.arccos([1, -1])
array([ 0.          ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```

larray.arctan

```
larray.arctan(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None,
              subok=True[, signature, extobj])
```

Trigonometric inverse tangent, element-wise.

larray specific variant of `numpy.arctan`.

Documentation from `numpy`:

The inverse of `tan`, so that if $y = \tan(x)$ then $x = \arctan(y)$.

Parameters

x [array_like]

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Out has the same shape as *x*. Its real part is in $[-\pi/2, \pi/2]$ (`arctan(+/-inf)` returns $\pm\pi/2$). This is a scalar if *x* is a scalar.

See also:

arctan2 The “four quadrant” arctan of the angle formed by (*x*, *y*) and the positive *x*-axis.

angle Argument of complex values.

Notes

arctan is a multi-valued function: for each *x* there are infinitely many numbers *z* such that $\tan(z) = x$. The convention is to return the angle *z* whose real part lies in $[-\pi/2, \pi/2]$.

For real-valued input data types, *arctan* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctan* is a complex analytic function that has $[1j, \text{infj}]$ and $[-1j, -\text{infj}]$ as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as *atan* or \tan^{-1} .

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79. <http://www.math.sfu.ca/~cbm/aands/>

Examples

We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```
>>> np.arctan([0, 1])
array([ 0.          ,  0.78539816])
```

```
>>> np.pi/4
0.78539816339744828
```

Plot arctan:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
>>> plt.axis('tight')
>>> plt.show()
```

larray.hypot

`larray.hypot(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Given the “legs” of a right triangle, return its hypotenuse.

larray specific variant of `numpy.hypot`.

Documentation from `numpy`:

Equivalent to `sqrt(x1**2 + x2**2)`, element-wise. If `x1` or `x2` is `scalar_like` (i.e., unambiguously castable to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

Parameters

x1, x2 [array_like] Leg of the triangle(s). If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is `True`, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

z [ndarray] The hypotenuse of the triangle(s). This is a scalar if both `x1` and `x2` are scalars.

Examples

```
>>> np.hypot(3*np.ones((3, 3)), 4*np.ones((3, 3)))
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

Example showing broadcast of scalar_like argument:

```
>>> np.hypot(3*np.ones((3, 3)), [4])
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

larray.arctan2

`larray.arctan2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.

larray specific variant of `numpy.arctan2`.

Documentation from `numpy`:

The quadrant (i.e., branch) is chosen so that `arctan2(x1, x2)` is the signed angle in radians between the ray ending at the origin and passing through the point (1,0), and the ray ending at the origin and passing through the point (x2, x1). (Note the role reversal: the “y-coordinate” is the first function parameter, the “x-coordinate” is the second.) By IEEE convention, this function is defined for $x2 = +/-0$ and for either or both of $x1$ and $x2 = +/-inf$ (see Notes for specific values).

This function is not defined for complex-valued arguments; for the so-called argument of complex values, use *angle*.

Parameters

x1 [array_like, real-valued] y-coordinates.

x2 [array_like, real-valued] x-coordinates. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

angle [ndarray] Array of angles in radians, in the range `[-pi, pi]`. This is a scalar if both *x1* and *x2* are scalars.

See also:

arctan, tan, angle

Notes

arctan2 is identical to the *atan2* function of the underlying C library. The following special values are defined in the C standard: [1]

<i>x1</i>	<i>x2</i>	<i>arctan2(x1,x2)</i>
+/- 0	+0	+/- 0
+/- 0	-0	+/- pi
> 0	+/-inf	+0 / +pi
< 0	+/-inf	-0 / -pi
+/-inf	+inf	+/- (pi/4)
+/-inf	-inf	+/- (3*pi/4)

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.

References

[1]

Examples

Consider four points in different quadrants:

```
>>> x = np.array([-1, +1, +1, -1])
>>> y = np.array([-1, -1, +1, +1])
>>> np.arctan2(y, x) * 180 / np.pi
array([-135., -45., 45., 135.] )
```

Note the order of the parameters. *arctan2* is defined also when *x2* = 0 and at several other special points, obtaining values in the range [-pi, pi]:

```
>>> np.arctan2([1., -1.], [0., 0.])
array([ 1.57079633, -1.57079633])
>>> np.arctan2([0., 0., np.inf], [+0., -0., np.inf])
array([ 0., 3.14159265, 0.78539816])
```

larray.degrees

`larray.degrees(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Convert angles from radians to degrees.

larray specific variant of `numpy.degrees`.

Documentation from `numpy`:

Parameters

x [array_like] Input array in radians.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray of floats] The corresponding degree values; if *out* was supplied this is a reference to it. This is a scalar if *x* is a scalar.

See also:

rad2deg equivalent function

Examples

Convert a radian array to degrees

```
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([  0.,  30.,  60.,  90., 120., 150., 180., 210., 240.,
        270., 300., 330.] )
```

```
>>> out = np.zeros((rad.shape))
>>> r = np.degrees(rad, out)
>>> np.all(r == out)
True
```

larray.radians

`larray.radians(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Convert angles from degrees to radians.

larray specific variant of `numpy.radians`.

Documentation from `numpy`:

Parameters

x [array_like] Input array in degrees.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the

default `out=None`, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The corresponding radian values. This is a scalar if *x* is a scalar.

See also:

deg2rad equivalent function

Examples

Convert a degree array to radians

```
>>> deg = np.arange(12.) * 30.  
>>> np.radians(deg)  
array([ 0.          ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,  
        2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,  
        5.23598776,  5.75958653])
```

```
>>> out = np.zeros((deg.shape))  
>>> ret = np.radians(deg, out)  
>>> ret is out  
True
```

larray.unwrap

`larray.unwrap(*args, **kwargs)`

Unwrap by changing deltas between values to 2π complement.

`larray` specific variant of `numpy.unwrap`.

Documentation from `numpy`:

Unwrap radian phase *p* by changing absolute jumps greater than *discont* to their 2π complement along the given axis.

Parameters

p [array_like] Input array.

discont [float, optional] Maximum discontinuity between values, default is `pi`.

axis [int, optional] Axis along which unwrap will operate, default is the last axis.

Returns

out [ndarray] Output array.

See also:

rad2deg, deg2rad

Notes

If the discontinuity in p is smaller than π , but larger than *discont*, no unwrapping is done because taking the 2π complement would only make the discontinuity larger.

Examples

```
>>> phase = np.linspace(0, np.pi, num=5)
>>> phase[3:] += np.pi
>>> phase
array([ 0.          ,  0.78539816,  1.57079633,  5.49778714,  6.28318531]) # may
↪ vary
>>> np.unwrap(phase)
array([ 0.          ,  0.78539816,  1.57079633, -0.78539816,  0.          ]) # may
↪ vary
```

Hyperbolic functions

<code>sinh(x, /[, out, where, casting, order, ...])</code>	Hyperbolic sine, element-wise.
<code>cosh(x, /[, out, where, casting, order, ...])</code>	Hyperbolic cosine, element-wise.
<code>tanh(x, /[, out, where, casting, order, ...])</code>	Compute hyperbolic tangent element-wise.
<code>arcsinh(x, /[, out, where, casting, order, ...])</code>	Inverse hyperbolic sine element-wise.
<code>arccosh(x, /[, out, where, casting, order, ...])</code>	Inverse hyperbolic cosine, element-wise.
<code>arctanh(x, /[, out, where, casting, order, ...])</code>	Inverse hyperbolic tangent element-wise.

larray.sinh

`larray.sinh(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Hyperbolic sine, element-wise.

larray specific variant of `numpy.sinh`.

Documentation from `numpy`:

Equivalent to $1/2 * (\text{np.exp}(x) - \text{np.exp}(-x))$ or $-1j * \text{np.sin}(1j*x)$.

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The corresponding hyperbolic sine values. This is a scalar if *x* is a scalar.

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83.

Examples

```
>>> np.sinh(0)
0.0
>>> np.sinh(np.pi*1j/2)
1j
>>> np.sinh(np.pi*1j) # (exact value is 0)
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.
```

```
>>> # Example of providing the optional output parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.sinh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.sinh(np.zeros((3,3)), np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

larray.cosh

`larray.cosh(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Hyperbolic cosine, element-wise.

larray specific variant of `numpy.cosh`.

Documentation from `numpy`:

Equivalent to $1/2 * (np.exp(x) + np.exp(-x))$ and `np.cos(1j*x)`.

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out*=None, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Output array of same shape as *x*. This is a scalar if *x* is a scalar.

Examples

```
>>> np.cosh(0)
1.0
```

The hyperbolic cosine describes the shape of a hanging cable:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 1000)
>>> plt.plot(x, np.cosh(x))
>>> plt.show()
```

larray.tanh

`larray.tanh(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Compute hyperbolic tangent element-wise.

larray specific variant of `numpy.tanh`.

Documentation from `numpy`:

Equivalent to `np.sinh(x) / np.cosh(x)` or `-1j * np.tan(1j*x)`.

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out*=None, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The corresponding hyperbolic tangent values. This is a scalar if *x* is a scalar.

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

[1], [2]

Examples

```
>>> np.tanh((0, np.pi*1j, np.pi*1j/2))
array([ 0. +0.00000000e+00j,  0. -1.22460635e-16j,  0. +1.63317787e+16j])
```

```
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.tanh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.tanh(np.zeros((3,3)), np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

larray.arcsinh

`larray.arcsinh`(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*,
subok=True, *signature*, *extobj*)

Inverse hyperbolic sine element-wise.

`larray` specific variant of `numpy.arcsinh`.

Documentation from `numpy`:

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Array of the same shape as *x*. This is a scalar if *x* is a scalar.

Notes

arcsinh is a multivalued function: for each x there are infinitely many numbers z such that $\sinh(z) = x$. The convention is to return the z whose imaginary part lies in $[-\pi/2, \pi/2]$.

For real-valued input data types, *arcsinh* always returns real output. For each value that cannot be expressed as a real number or infinity, it returns `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytical function that has branch cuts $[1j, infj]$ and $[-1j, -infj]$ and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as *asinh* or \sinh^{-1} .

References

[1], [2]

Examples

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])
```

larray.arccosh

`larray.arccosh(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Inverse hyperbolic cosine, element-wise.

`larray` specific variant of `numpy.arccosh`.

Documentation from `numpy`:

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is `True`, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

arccosh [ndarray] Array of the same shape as `x`. This is a scalar if `x` is a scalar.

See also:

[*cosh*](#), [*arcsinh*](#), [*sinh*](#), [*arctanh*](#), [*tanh*](#)

Notes

arccosh is a multivalued function: for each x there are infinitely many numbers z such that $\cosh(z) = x$. The convention is to return the z whose imaginary part lies in $[-\pi, \pi]$ and the real part in $[0, \infty]$.

For real-valued input data types, *arccosh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccosh* is a complex analytical function that has a branch cut $[-\infty, 1]$ and is continuous from above on it.

References

[1], [2]

Examples

```
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])
>>> np.arccosh(1)
0.0
```

larray.arctanh

`larray.arctanh`(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*, *signature*, *extobj*)

Inverse hyperbolic tangent element-wise.

`larray` specific variant of `numpy.arctanh`.

Documentation from `numpy`:

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is `True`, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Array of the same shape as *x*. This is a scalar if *x* is a scalar.

See also:

`emath.arctanh`

Notes

arctanh is a multivalued function: for each x there are infinitely many numbers z such that $\tanh(z) = x$. The convention is to return the z whose imaginary part lies in $[-\pi/2, \pi/2]$.

For real-valued input data types, *arctanh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctanh* is a complex analytical function that has branch cuts $[-1, -\infty]$ and $[1, \infty]$ and is continuous from above on the former and from below on the latter.

The inverse hyperbolic tangent is also known as *atanh* or \tanh^{-1} .

References

[1], [2]

Examples

```
>>> np.arctanh([0, -0.5])
array([ 0.          , -0.54930614])
```

Complex Numbers

<i>angle</i> (*args, **kwargs)	Return the angle of the complex argument.
<i>real</i> (*args, **kwargs)	Return the real part of the complex argument.
<i>imag</i> (*args, **kwargs)	Return the imaginary part of the complex argument.
<i>conj</i> (x, /[, out, where, casting, order, ...])	Return the complex conjugate, element-wise.

larray.angle

`larray.angle(*args, **kwargs)`

Return the angle of the complex argument.

larray specific variant of `numpy.angle`.

Documentation from `numpy`:

Parameters

z [array_like] A complex number or sequence of complex numbers.

deg [bool, optional] Return angle in degrees if True, radians if False (default).

Returns

angle [ndarray or scalar] The counterclockwise angle from the positive real axis on the complex plane in the range $(-\pi, \pi]$, with dtype as `numpy.float64`.

..versionchanged:: 1.16.0 This function works on subclasses of `ndarray` like `ma.array`.

See also:

[*arctan2*](#)

absolute

Examples

```
>>> np.angle([1.0, 1.0j, 1+1j])           # in radians
array([ 0.          ,  1.57079633,  0.78539816]) # may vary
>>> np.angle(1+1j, deg=True)               # in degrees
45.0
```

larray.real

`larray.real(*args, **kwargs)`

Return the real part of the complex argument.

larray specific variant of `numpy.real`.

Documentation from `numpy`:

Parameters

val [array_like] Input array.

Returns

out [ndarray or scalar] The real component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

See also:

`real_if_close`, `imag`, `angle`

Examples

```
>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.real
array([1.,  3.,  5.])
>>> a.real = 9
>>> a
array([9.+2.j,  9.+4.j,  9.+6.j])
>>> a.real = np.array([9, 8, 7])
>>> a
array([9.+2.j,  8.+4.j,  7.+6.j])
>>> np.real(1 + 1j)
1.0
```

larray.imag

`larray.imag(*args, **kwargs)`

Return the imaginary part of the complex argument.

`larray` specific variant of `numpy.imag`.

Documentation from `numpy`:

Parameters

val [array_like] Input array.

Returns

out [ndarray or scalar] The imaginary component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

See also:

[`real`](#), [`angle`](#), [`real_if_close`](#)

Examples

```

>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.imag
array([2., 4., 6.])
>>> a.imag = np.array([8, 10, 12])
>>> a
array([1. +8.j, 3.+10.j, 5.+12.j])
>>> np.imag(1 + 1j)
1.0

```

larray.conj

`larray.conj(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Return the complex conjugate, element-wise.

`larray` specific variant of `numpy.conjugate`.

Documentation from `numpy`:

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Parameters

x [array_like] Input value.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The complex conjugate of *x*, with same dtype as *y*. This is a scalar if *x* is a scalar.

Notes

conj is an alias for *conjugate*:

```
>>> np.conj is np.conjugate
True
```

Examples

```
>>> np.conjugate(1+2j)
(1-2j)
```

```
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j,  0.-0.j],
       [ 0.-0.j,  1.-1.j]])
```

Floating Point Routines

<i>signbit</i> (x, /, out, where, casting, order, ...)	Returns element-wise True where signbit is set (less than zero).
<i>copysign</i> (x1, x2, /, out, where, casting, ...)	Change the sign of x1 to that of x2, element-wise.
<i>frexp</i> (x[, out1, out2], / [[, out, where, ...])	Decompose the elements of x into mantissa and twos exponent.
<i>ldexp</i> (x1, x2, /, out, where, casting, ...)	Returns $x1 * 2^{x2}$, element-wise.

larray.signbit

`larray.signbit(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Returns element-wise True where signbit is set (less than zero).

larray specific variant of `numpy.signbit`.

Documentation from `numpy`:

Parameters

x [array_like] The input value(s).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out*

array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

result [ndarray of bool] Output array, or reference to *out* if that was supplied. This is a scalar if *x* is a scalar.

Examples

```
>>> np.signbit(-1.2)
True
>>> np.signbit(np.array([1, -2.3, 2.1]))
array([False,  True, False])
```

larray.copysign

`larray.copysign(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Change the sign of *x1* to that of *x2*, element-wise.

larray specific variant of `numpy.copysign`.

Documentation from `numpy`:

If *x2* is a scalar, its sign will be copied to all elements of *x1*.

Parameters

x1 [array_like] Values to change the sign of.

x2 [array_like] The sign of *x2* is copied to *x1*. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] The values of *x1* with the sign of *x2*. This is a scalar if both *x1* and *x2* are scalars.

Examples

```
>>> np.copysign(1.3, -1)
-1.3
>>> 1/np.copysign(0, 1)
inf
>>> 1/np.copysign(0, -1)
-inf
```

```
>>> np.copysign([-1, 0, 1], -1.1)
array([-1., -0., -1.])
>>> np.copysign([-1, 0, 1], np.arange(3)-1)
array([-1., 0., 1.])
```

larray.frexp

`larray.frexp(x[, out1, out2], /[, out=(None, None)], *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Decompose the elements of `x` into mantissa and twos exponent.

`larray` specific variant of `numpy.frexp`.

Documentation from `numpy`:

Returns (*mantissa*, *exponent*), where $x = \text{mantissa} * 2^{**\text{exponent}}$. The mantissa is lies in the open interval(-1, 1), while the twos exponent is a signed integer.

Parameters

x [array_like] Array of numbers to be decomposed.

out1 [ndarray, optional] Output array for the mantissa. Must have the same shape as `x`.

out2 [ndarray, optional] Output array for the exponent. Must have the same shape as `x`.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

mantissa [ndarray] Floating values between -1 and 1. This is a scalar if `x` is a scalar.

exponent [ndarray] Integer exponents of 2. This is a scalar if `x` is a scalar.

See also:

ldexp Compute $y = x1 * 2^{**x2}$, the inverse of *frexp*.

Notes

Complex dtypes are not supported, they will raise a `TypeError`.

Examples

```
>>> x = np.arange(9)
>>> y1, y2 = np.frexp(x)
>>> y1
array([ 0.    ,  0.5   ,  0.5   ,  0.75  ,  0.5   ,  0.625 ,  0.75  ,  0.875 ,
        0.5   ])
>>> y2
array([0, 1, 2, 2, 3, 3, 3, 3, 4])
>>> y1 * 2**y2
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.] )
```

larray.ldexp

`larray.ldexp(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])`

Returns $x1 * 2^{x2}$, element-wise.

larray specific variant of `numpy.ldexp`.

Documentation from `numpy`:

The mantissas $x1$ and twos exponents $x2$ are used to construct floating point numbers $x1 * 2^{x2}$.

Parameters

x1 [array_like] Array of multipliers.

x2 [array_like, int] Array of twos exponents. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] This condition is broadcast over the input. At locations where the condition is `True`, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is `False` will remain uninitialized.

****kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The result of $x1 * 2^{x2}$. This is a scalar if both $x1$ and $x2$ are scalars.

See also:

[`frexp`](#) Return $(y1, y2)$ from $x = y1 * 2^{y2}$, inverse to `ldexp`.

Notes

Complex dtypes are not supported, they will raise a `TypeError`.

`ldexp` is useful as the inverse of `frexp`, if used by itself it is more clear to simply use the expression `x1 * 2**x2`.

Examples

```
>>> np.ldexp(5, np.arange(4))
array([ 5., 10., 20., 40.], dtype=float16)
```

```
>>> x = np.arange(6)
>>> np.ldexp(*np.frexp(x))
array([ 0.,  1.,  2.,  3.,  4.,  5.]
```

4.3.7 Metadata

Metadata

An ordered dictionary allowing key-values accessibly using attribute notation (`AttributeDict.attribute`) instead of key notation (`Dict["key"]`).

`larray.Metadata`

class `larray.Metadata`

An ordered dictionary allowing key-values accessibly using attribute notation (`AttributeDict.attribute`) instead of key notation (`Dict["key"]`).

Examples

```
>>> from larray import ndtest
>>> from datetime import datetime
```

Add metadata at array initialization

```
>>> # Python 2 or <= 3.5
>>> arr = ndtest((3, 3), meta=[('title', 'the title'), ('author', 'John Smith')])
>>> # Python 3.6+
>>> arr = ndtest((3, 3), meta=Metadata(title='the title', author='John Smith'))
```

Add metadata after array initialization

```
>>> arr.meta.creation_date = datetime(2017, 2, 10)
```

Access to metadata

```
>>> arr.meta.creation_date
datetime.datetime(2017, 2, 10, 0, 0)
```

Modify metadata


```
>>> arr.meta.creation_date = datetime(2017, 2, 16)
```

Delete metadata

```
>>> del arr.meta.creation_date
```

`__init__(self, /, *args, **kwargs)`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self, /, *args, **kwargs)</code>	Initialize self.
<code>clear()</code>	
<code>copy()</code>	
<code>from_array(array)</code>	
<code>from_hdf(hdfstore[, key])</code>	
<code>fromkeys()</code>	If not specified, the value defaults to None.
<code>get()</code>	
<code>items()</code>	
<code>keys()</code>	
<code>move_to_end()</code>	Move an existing element to the end (or beginning if <code>last==False</code>).
<code>pop()</code>	value.
<code>popitem(self, /[, last])</code>	Remove and return a (key, value) pair from the dictionary.
<code>setdefault()</code>	
<code>to_hdf(self, hdfstore[, key])</code>	
<code>update()</code>	If E is present and has a <code>.keys()</code> method, then does: for k in E: D[k] = E[k] If E is present and lacks a <code>.keys()</code> method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
<code>values()</code>	

4.3.8 Input/Output

Read

<code>read_csv(filepath_or_buffer[, nb_axes, ...])</code>	Reads csv file and returns an array with the contents.
<code>read_tsv(filepath_or_buffer, **kwargs)</code>	
<code>read_excel(filepath[, sheet, nb_axes, ...])</code>	Reads excel file from sheet name and returns an Array with the contents
<code>read_hdf(filepath_or_buffer, key[, ...])</code>	Reads an axis or group or array named key from a HDF5 file in filepath (path+name)
<code>read_eurostat(filepath_or_buffer, **kwargs)</code>	Reads EUROSTAT TSV (tab-separated) file into an array.
<code>read_sas(filepath[, nb_axes, index_col, ...])</code>	Reads sas file and returns an Array with the contents

Continued on next page

Table 52 – continued from previous page

<code>read_stata(filepath_or_buffer[, index_col, ...])</code>	Reads Stata .dta file and returns an Array with the contents
---	--

larray.read_csv

`larray.read_csv(filepath_or_buffer, nb_axes=None, index_col=None, sep=', ', headersep=None, fill_value=nan, na=nan, sort_rows=False, sort_columns=False, wide=True, dialect='larray', **kwargs)`

Reads csv file and returns an array with the contents.

Parameters

filepath_or_buffer [str or any file-like object] Path where the csv file has to be read or a file handle.

nb_axes [int or None, optional] Number of axes of output array. The first `nb_axes - 1` columns and the header of the CSV file will be used to set the axes of the output array. If not specified, the number of axes is given by the position of the first column header including a `\` character plus one. If no column header includes a `\` character, the array is assumed to have one axis. Defaults to None.

index_col [list or None, optional] Positions of columns for the `n-1` first axes (ex. `[0, 1, 2, 3]`). Defaults to None (see `nb_axes` above).

sep [str, optional] Separator.

headersep [str or None, optional] Separator for headers.

fill_value [scalar or Array, optional] Value used to fill cells corresponding to label combinations which are not present in the input. Defaults to NaN.

sort_rows [bool, optional] Whether or not to sort the rows alphabetically (sorting is more efficient than not sorting). Defaults to False.

sort_columns [bool, optional] Whether or not to sort the columns alphabetically (sorting is more efficient than not sorting). Defaults to False.

wide [bool, optional] Whether or not to assume the array is stored in “wide” format. If False, the array is assumed to be stored in “narrow” format: one column per axis plus one value column. Defaults to True.

dialect [{ 'classic', 'larray', 'liam2' }, optional] Name of dialect. Defaults to 'larray'.

****kwargs** Extra keyword arguments are passed on to `pandas.read_csv`

Returns

Array

Notes

Without using any argument to tell otherwise, the csv files are assumed to be in this format:

```
axis0_name,axis1_name\axis2_name,axis2_label0,axis2_label1
axis0_label0,axis1_label0,value,value
axis0_label0,axis1_label1,value,value
axis0_label1,axis1_label0,value,value
axis0_label1,axis1_label1,value,value
```

For example:

```
country,gender\time,2013,2014,2015
Belgium,Male,5472856,5493792,5524068
Belgium,Female,5665118,5687048,5713206
France,Male,31772665,32045129,32174258
France,Female,33827685,34120851,34283895
Germany,Male,39380976,39556923,39835457
Germany,Female,41142770,41210540,41362080
```

Examples

```
>>> csv_dir = get_example_filepath('examples')
>>> fname = csv_dir + '/pop.csv'
```

```
>>> # The data below is derived from a subset of the demo_pjan table from Eurostat
>>> read_csv(fname)
country gender\time      2013      2014      2015
Belgium      Male    5472856    5493792    5524068
Belgium      Female  5665118    5687048    5713206
France      Male    31772665   32045129   32174258
France      Female  33827685   34120851   34283895
Germany      Male    39380976   39556923   39835457
Germany      Female  41142770   41210540   41362080
```

Missing label combinations

```
>>> fname = csv_dir + '/pop_missing_values.csv'
>>> # let's take a look inside the CSV file.
>>> # they are missing label combinations: (Paris, male) and (New York, female)
>>> with open(fname) as f:
...     print(f.read().strip())
country,gender\time,2013,2014,2015
Belgium,Male,5472856,5493792,5524068
Belgium,Female,5665118,5687048,5713206
France,Female,33827685,34120851,34283895
Germany,Male,39380976,39556923,39835457
>>> # by default, cells associated with missing label combinations are filled
↳with NaN.
>>> # In that case, an int array is converted to a float array.
>>> read_csv(fname)
country gender\time      2013      2014      2015
Belgium      Male    5472856.0    5493792.0    5524068.0
Belgium      Female  5665118.0    5687048.0    5713206.0
France      Male              nan              nan              nan
```

(continues on next page)

(continued from previous page)

```

France      Female  33827685.0  34120851.0  34283895.0
Germany     Male    39380976.0  39556923.0  39835457.0
Germany     Female      nan      nan      nan
>>> # using argument 'fill_value', you can choose which value to fill_
    ↪missing cells.
>>> read_csv(fname, fill_value=0)
country  gender\time      2013      2014      2015
Belgium   Male    5472856    5493792    5524068
Belgium   Female  5665118    5687048    5713206
France    Male      0         0         0
France    Female  33827685    34120851    34283895
Germany   Male    39380976    39556923    39835457
Germany   Female      0         0         0

```

Specify the number of axes of the output array (useful when the name of the last axis is implicit)

```

>>> fname = csv_dir + '/pop_missing_axis_name.csv'
>>> # let's take a look inside the CSV file.
>>> # The name of the last axis is missing.
>>> with open(fname) as f:
...     print(f.read().strip())
country,gender,2013,2014,2015
Belgium,Male,5472856,5493792,5524068
Belgium,Female,5665118,5687048,5713206
France,Male,31772665,32045129,32174258
France,Female,33827685,34120851,34283895
Germany,Male,39380976,39556923,39835457
Germany,Female,41142770,41210540,41362080
>>> # read the array stored in the CSV file as is
>>> arr = read_csv(fname)
>>> # we expected a 3 x 2 x 3 array with data of type int
>>> # but we got a 6 x 4 array with data of type object
>>> arr.info
6 x 4
country [6]: 'Belgium' 'Belgium' 'France' 'France' 'Germany' 'Germany'
{1} [4]: 'gender' '2013' '2014' '2015'
dtype: object
memory used: 192 bytes
>>> # using argument 'nb_axes', you can force the number of axes of the output_
    ↪array
>>> arr = read_csv(fname, nb_axes=3)
>>> # as expected, we have a 3 x 2 x 3 array with data of type int
>>> arr.info
3 x 2 x 3
country [3]: 'Belgium' 'France' 'Germany'
gender [2]: 'Male' 'Female'
{2} [3]: 2013 2014 2015
dtype: int64
memory used: 144 bytes

```

Read array saved in “narrow” format (wide=False)

```

>>> fname = csv_dir + '/pop_narrow_format.csv'
>>> # let's take a look inside the CSV file.
>>> # Here, data are stored in a 'narrow' format.
>>> with open(fname) as f:
...     print(f.read().strip())

```

(continues on next page)

(continued from previous page)

```
country,time,value
Belgium,2013,11137974
Belgium,2014,11180840
Belgium,2015,11237274
France,2013,65600350
France,2014,66165980
France,2015,66458153
>>> # to read arrays stored in 'narrow' format, you must pass wide=False to read_
↳ CSV
>>> read_csv(fname, wide=False)
country\time      2013      2014      2015
    Belgium  11137974  11180840  11237274
    France   65600350   66165980   66458153
```

larray.read_tsv

`larray.read_tsv(filepath_or_buffer, **kwargs)`

larray.read_excel

`larray.read_excel(filepath, sheet=0, nb_axes=None, index_col=None, fill_value=nan, na=nan, sort_rows=False, sort_columns=False, wide=True, engine=None, range=slice(None, None, None), **kwargs)`

Reads excel file from sheet name and returns an Array with the contents

Parameters

filepath [str] Path where the Excel file has to be read or use -1 to refer to the currently active workbook.

sheet [str, Group or int, optional] Name or index of the Excel sheet containing the array to be read. By default the array is read from the first sheet.

nb_axes [int, optional] Number of axes of output array. The first `nb_axes - 1` columns and the header of the Excel sheet will be used to set the axes of the output array. If not specified, the number of axes is given by the position of the first column header including a `\` character plus one. If no column header includes a `\` character, the array is assumed to have one axis. Defaults to None.

index_col [list, optional] Positions of columns for the n-1 first axes (ex. [0, 1, 2, 3]). Defaults to None (see `nb_axes` above).

fill_value [scalar or Array, optional] Value used to fill cells corresponding to label combinations which are not present in the input. Defaults to NaN.

sort_rows [bool, optional] Whether or not to sort the rows alphabetically (sorting is more efficient than not sorting). Defaults to False.

sort_columns [bool, optional] Whether or not to sort the columns alphabetically (sorting is more efficient than not sorting). Defaults to False.

wide [bool, optional] Whether or not to assume the array is stored in “wide” format. If False, the array is assumed to be stored in “narrow” format: one column per axis plus one value column. Defaults to True.

engine [{‘xlrd’, ‘xlwings’}, optional] Engine to use to read the Excel file. If None (default), it will use ‘xlwings’ by default if the module is installed and relies on Pandas default reader otherwise.

range [str, optional] Range to load the array from (only supported for the ‘xlwings’ engine). Defaults to slice(None) which loads the whole sheet, ignoring blank cells in the bottom right corner.

****kwargs**

Returns

Array

Examples

```
>>> fname = get_example_filepath('examples.xlsx')
```

Read array from first sheet

```
>>> # The data below is derived from a subset of the demo_pjan table from Eurostat
>>> read_excel(fname)
country gender\time      2013      2014      2015
Belgium      Male    5472856    5493792    5524068
Belgium      Female  5665118    5687048    5713206
France       Male    31772665   32045129   32174258
France       Female  33827685   34120851   34283895
Germany      Male    39380976   39556923   39835457
Germany      Female  41142770   41210540   41362080
```

Read array from a specific sheet

```
>>> # The data below is derived from a subset of the demo_fasec table from
↳Eurostat
>>> read_excel(fname, 'births')
country gender\time      2013      2014      2015
Belgium      Male    64371     64173     62561
Belgium      Female  61235     60841     59713
France       Male    415762    418721    409145
France       Female  396581    400607    390526
Germany      Male    349820    366835    378478
Germany      Female  332249    348092    359097
```

Missing label combinations

Let us take a look inside the sheet ‘pop_missing_values’. Note the missing label combinations: (Paris, male) and (New York, female):

```
country gender\time      2013      2014      2015
Belgium      Male    5472856    5493792    5524068
Belgium      Female  5665118    5687048    5713206
France       Female  33827685   34120851   34283895
Germany      Male    39380976   39556923   39835457
```

By default, cells associated with missing label combinations are filled with NaN. In that case, an int array is converted to a float array.

```
>>> read_excel(fname, sheet='pop_missing_values')
country gender\time      2013      2014      2015
Belgium      Male    5472856.0    5493792.0    5524068.0
Belgium      Female  5665118.0    5687048.0    5713206.0
France       Male          nan          nan          nan
France       Female  33827685.0    34120851.0    34283895.0
Germany      Male    39380976.0    39556923.0    39835457.0
Germany      Female          nan          nan          nan
```

Using the `fill_value` argument, you can choose another value to use to fill missing cells.

```
>>> read_excel(fname, sheet='pop_missing_values', fill_value=0)
country gender\time      2013      2014      2015
Belgium      Male    5472856    5493792    5524068
Belgium      Female  5665118    5687048    5713206
France       Male          0          0          0
France       Female  33827685    34120851    34283895
Germany      Male    39380976    39556923    39835457
Germany      Female          0          0          0
```

Specify the number of axes of the output array (useful when the name of the last axis is implicit)

The content of the sheet ‘missing_axis_name’ is:

```
country gender      2013      2014      2015
Belgium Male    5472856    5493792    5524068
Belgium Female  5665118    5687048    5713206
France Male    31772665    32045129    32174258
France Female  33827685    34120851    34283895
Germany Male    39380976    39556923    39835457
Germany Female  41142770    41210540    41362080
```

```
>>> # read the array stored in the sheet 'pop_missing_axis_name' as is
>>> arr = read_excel(fname, sheet='pop_missing_axis_name')
>>> # we expected a 3 x 2 x 3 array with data of type int
>>> # but we got a 6 x 4 array with data of type object
>>> arr.info
6 x 4
country [6]: 'Belgium' 'Belgium' 'France' 'France' 'Germany' 'Germany'
{1} [4]: 'gender' '2013' '2014' '2015'
dtype: object
memory used: 192 bytes
>>> # using argument 'nb_axes', you can force the number of axes of the output_
↳array
>>> arr = read_excel(fname, sheet='pop_missing_axis_name', nb_axes=3)
>>> # as expected, we have a 3 x 2 x 3 array with data of type int
>>> arr.info
3 x 2 x 3
country [3]: 'Belgium' 'France' 'Germany'
gender [2]: 'Male' 'Female'
{2} [3]: 2013 2014 2015
dtype: int64
memory used: 144 bytes
```

Read array saved in “narrow” format (`wide=False`)

Let us take a look inside the sheet ‘pop_narrow’ where the data is stored in a ‘narrow’ format:

country	time	value
Belgium	2013	11137974
Belgium	2014	11180840
Belgium	2015	11237274
France	2013	65600350
France	2014	66165980
France	2015	66458153

```
>>> # to read arrays stored in 'narrow' format, you must pass wide=False to read_
↳ excel
>>> read_excel(fname, 'pop_narrow_format', wide=False)
country\time      2013      2014      2015
Belgium  11137974  11180840  11237274
France   65600350  66165980  66458153
```

Extract array from a given range (xlwings only)

```
>>> read_excel(fname, 'pop_births_deaths', range='A9:E15')
country gender\time  2013  2014  2015
Belgium      Male   64371  64173  62561
Belgium      Female 61235  60841  59713
France      Male  415762  418721  409145
France      Female 396581  400607  390526
Germany      Male  349820  366835  378478
Germany      Female 332249  348092  359097
```

larray.read_hdf

`larray.read_hdf` (filepath_or_buffer, key, fill_value=nan, na=nan, sort_rows=False, sort_columns=False, name=None, **kwargs)

Reads an axis or group or array named key from a HDF5 file in filepath (path+name)

Parameters

filepath_or_buffer [str or pandas.HDFStore] Path and name where the HDF5 file is stored or a HDFStore object.

key [str or Group] Name of the array.

fill_value [scalar or Array, optional] Value used to fill cells corresponding to label combinations which are not present in the input. Defaults to NaN.

sort_rows [bool, optional] Whether or not to sort the rows alphabetically. Must be False if the read array has been dumped with an larray version ≥ 0.30 . Defaults to False.

sort_columns [bool, optional] Whether or not to sort the columns alphabetically. Must be False if the read array has been dumped with an larray version ≥ 0.30 . Defaults to False.

name [str, optional] Name of the axis or group to return. If None, name is set to passed key. Defaults to None.

Returns

Array

Examples

```
>>> fname = get_example_filepath('examples.h5')
```

Read array by passing its identifier (key) inside the HDF file

```
>>> # The data below is derived from a subset of the demo_pjan table from Eurostat
>>> read_hdf(fname, 'pop')
country gender\time      2013      2014      2015
Belgium      Male    5472856    5493792    5524068
Belgium      Female  5665118    5687048    5713206
France       Male    31772665   32045129   32174258
France       Female  33827685   34120851   34283895
Germany      Male    39380976   39556923   39835457
Germany      Female  41142770   41210540   41362080
```

larray.read_eurostat

`larray.read_eurostat` (*filepath_or_buffer*, ***kwargs*)

Reads EUROSTAT TSV (tab-separated) file into an array.

EUROSTAT TSV files are special because they use tabs as data separators but comas to separate headers.

Parameters

filepath_or_buffer [str or any file-like object] Path where the tsv file has to be read or a file handle.

kwargs Arbitrary keyword arguments are passed through to `read_csv`.

Returns

Array

larray.read_sas

`larray.read_sas` (*filepath*, *nb_axes=None*, *index_col=None*, *fill_value=nan*, *na=nan*, *sort_rows=False*, *sort_columns=False*, ***kwargs*)

Reads sas file and returns an Array with the contents *nb_axes*: number of axes of the output array

or *index_col*: Positions of columns for the n-1 first axes (ex. [0, 1, 2, 3])

larray.read_stata

`larray.read_stata` (*filepath_or_buffer*, *index_col=None*, *sort_rows=False*, *sort_columns=False*, ***kwargs*)

Reads Stata .dta file and returns an Array with the contents

Parameters

filepath_or_buffer [str or file-like object] Path to .dta file or a file handle.

index_col [str or None, optional] Name of column to set as index. Defaults to None.

sort_rows [bool, optional] Whether or not to sort the rows alphabetically (sorting is more efficient than not sorting). This only makes sense in combination with *index_col*. Defaults to False.

sort_columns [bool, optional] Whether or not to sort the columns alphabetically (sorting is more efficient than not sorting). Defaults to False.

Returns

Array

See also:

[*Array.to_stata*](#)

Notes

The round trip to Stata ([*Array.to_stata*](#) followed by [*read_stata*](#)) loose the name of the “column” axis.

Examples

```
>>> read_stata('test.dta')
{0}\{1}  row  country  sex
      0    0      BE    F
      1    1      FR    M
      2    2      FR    F
>>> read_stata('test.dta', index_col='row')
row\{1}  country  sex
      0      BE    F
      1      FR    M
      2      FR    F
```

Write

<i>Array.to_csv</i> (self, filepath[, sep, na_rep, ...])	Writes array to a csv file.
<i>Array.to_excel</i> (self[, filepath, sheet, ...])	Writes array in the specified sheet of specified excel workbook.
<i>Array.to_hdf</i> (self, filepath, key)	Writes array to a HDF file.
<i>Array.to_stata</i> (self, filepath_or_buffer, ...)	Writes array to a Stata .dta file.
<i>Array.dump</i> (self[, header, wide, value_name, ...])	Dump array as a 2D nested list.

[**larray.Array.to_csv**](#)

[*Array.to_csv*](#)(self, filepath, sep=', ', na_rep="", wide=True, value_name='value', dialect='default', **kwargs)

Writes array to a csv file.

Parameters

filepath [str] path where the csv file has to be written.

sep [str, optional] separator for the csv file. Defaults to ,.

na_rep [str, optional] replace NA values with na_rep. Defaults to ‘’.

wide [boolean, optional] Whether or not writing arrays in “wide” format. If True, arrays are exported with the last axis represented horizontally. If False, arrays are exported in “narrow” format: one column per axis plus one value column. Defaults to True.

value_name [str, optional] Name of the column containing the values (last column) in the csv file when *wide=False* (see above). Defaults to 'value'.

dialect ['default' | 'classic', optional] Whether or not to write the last axis name (using ``). Defaults to 'default'.

dropna [None, 'all', 'any' or True, optional] Drop lines if 'all' its values are NA, if 'any' value is NA or do not drop any line (default). True is equivalent to 'all'.

Examples

```
>>> tmpdir = getfixture('tmpdir')
>>> fname = os.path.join(tmpdir.strpath, 'test.csv')
>>> a = ndtest('nat=BE,FO;sex=M,F')
>>> a
nat\sex  M  F
      BE  0  1
      FO  2  3
>>> a.to_csv(fname)
>>> with open(fname) as f:
...     print(f.read().strip())
nat\sex,M,F
BE,0,1
FO,2,3
>>> a.to_csv(fname, sep=';', wide=False)
>>> with open(fname) as f:
...     print(f.read().strip())
nat;sex;value
BE;M;0
BE;F;1
FO;M;2
FO;F;3
>>> a.to_csv(fname, sep=';', wide=False, value_name='population')
>>> with open(fname) as f:
...     print(f.read().strip())
nat;sex;population
BE;M;0
BE;F;1
FO;M;2
FO;F;3
>>> a.to_csv(fname, dialect='classic')
>>> with open(fname) as f:
...     print(f.read().strip())
nat,M,F
BE,0,1
FO,2,3
```

larray.Array.to_excel

`Array.to_excel(self, filepath=None, sheet=None, position='A1', overwrite_file=False, clear_sheet=False, header=True, transpose=False, wide=True, value_name='value', engine=None, *args, **kwargs)`

Writes array in the specified sheet of specified excel workbook.

Parameters

filepath [str or int or None, optional] Path where the excel file has to be written. If None (default), creates a new Excel Workbook in a live Excel instance (Windows only). Use -1 to use the currently active Excel Workbook. Use a name without extension (.xlsx) to use any unsaved* workbook.

sheet [str or Group or int or None, optional] Sheet where the data has to be written. Defaults to None, Excel standard name if adding a sheet to an existing file, “Sheet1” otherwise. sheet can also refer to the position of the sheet (e.g. 0 for the first sheet, -1 for the last one).

position [str or tuple of integers, optional] Integer position (row, column) must be 1-based. Used only if engine is ‘xlwings’. Defaults to ‘A1’.

overwrite_file [bool, optional] Whether or not to overwrite the existing file (or just modify the specified sheet). Defaults to False.

clear_sheet [bool, optional] Whether or not to clear the existing sheet (if any) before writing. Defaults to False.

header [bool, optional] Whether or not to write a header (axes names and labels). Defaults to True.

transpose [bool, optional] Whether or not to transpose the array over last axis. This is equivalent to paste with option transpose in Excel. Defaults to False.

wide [boolean, optional] Whether or not writing arrays in “wide” format. If True, arrays are exported with the last axis represented horizontally. If False, arrays are exported in “narrow” format: one column per axis plus one value column. Defaults to True.

value_name [str, optional] Name of the column containing the values (last column) in the Excel sheet when *wide=False* (see above). Defaults to ‘value’.

engine [‘xlwings’ | ‘openpyxl’ | ‘xlsxwriter’ | ‘xlwt’ | None, optional] Engine to use to make the output. If None (default), it will use ‘xlwings’ by default if the module is installed and relies on Pandas default writer otherwise.

***args**

****kwargs**

Examples

```
>>> a = ndtest('nat=BE,FO;sex=M,F')
>>> # write to a new (unnamed) sheet
>>> a.to_excel('test.xlsx')
>>> # write to top-left corner of an existing sheet
>>> a.to_excel('test.xlsx', 'Sheet1')
>>> # add to existing sheet starting at position A15
>>> a.to_excel('test.xlsx', 'Sheet1', 'A15')
```

larray.Array.to_hdf

`Array.to_hdf(self, filepath, key)`

Writes array to a HDF file.

A HDF file can contain multiple arrays. The ‘key’ parameter is a unique identifier for the array.

Parameters

filepath [str] Path where the hdf file has to be written.

key [str or Group] Key (path) of the array within the HDF file (see Notes below).

Notes

Objects stored in a HDF file can be grouped together in *HDF groups*. If an object ‘my_obj’ is stored in a HDF group ‘my_group’, the key associated with this object is then ‘my_group/my_obj’. Be aware that a HDF group can have subgroups.

Examples

```
>>> a = ndtest((2, 3))
```

Save an array

```
>>> a.to_hdf('test.h5', 'a')
```

Save an array in a specific HDF group

```
>>> a.to_hdf('test.h5', 'arrays/a')
```

larray.Array.to_stata

`Array.to_stata(self, filepath_or_buffer, **kwargs)`

Writes array to a Stata .dta file.

Parameters

filepath_or_buffer [str or file-like object] Path to .dta file or a file handle.

See also:

[`read_stata`](#)

Notes

The round trip to Stata (`Array.to_stata` followed by `read_stata`) loses the name of the “column” axis.

Examples

```
>>> axes = [Axis(3, 'row'), Axis('column=country,sex')]
>>> arr = Array([[ 'BE', 'F'],
...             [ 'FR', 'M'],
...             [ 'FR', 'F']], axes=axes)
>>> arr
row*\column  age  sex
           0   5   F
           1  25   M
           2  30   F
>>> arr.to_stata('test.dta')
```

larray.Array.dump

`Array.dump(self, header=True, wide=True, value_name='value', light=False, axes_names=True, na_repr='as_is', maxlines=-1, edgeitems=5)`

Dump array as a 2D nested list. This is especially useful when writing to an Excel sheet via `open_excel()`.

Parameters

header [bool] Whether or not to output axes names and labels.

wide [boolean, optional] Whether or not to write arrays in “wide” format. If True, arrays are exported with the last axis represented horizontally. If False, arrays are exported in “narrow” format: one column per axis plus one value column. Not used if `header=False`. Defaults to True.

value_name [str, optional] Name of the column containing the values (last column) when `wide=False` (see above). Not used if `header=False`. Defaults to ‘value’.

light [bool, optional] Whether or not to hide repeated labels. In other words, only show a label if it is different from the previous one. Defaults to False.

axes_names [bool or ‘except_last’, optional] Assuming header is True, whether or not to include axes names. If `axes_names` is ‘except_last’, all axes names will be included except the last. Defaults to True.

na_repr [any scalar, optional] Replace missing values (NaN floats) by this value. Defaults to ‘as_is’ (do not do any replacement).

maxlines [int, optional] Maximum number of lines to show. Defaults to -1 (all lines are shown).

edgeitems [int, optional] If number of lines to display is greater than `maxlines`, only the first and last `edgeitems` lines are displayed. Only active if `maxlines` is not -1. Defaults to 5.

Returns

2D nested list or None for 0d arrays

Examples

```
>>> arr = ndtest((2, 2, 2))
>>> arr.dump()
[['a', 'b\\c', 'c0', 'c1'],
 ['a0', 'b0', 0, 1],
 ['a0', 'b1', 2, 3],
 ['a1', 'b0', 4, 5],
 ['a1', 'b1', 6, 7]]
>>> arr.dump(axes_names=False)
[['', '', 'c0', 'c1'],
 ['a0', 'b0', 0, 1],
 ['a0', 'b1', 2, 3],
 ['a1', 'b0', 4, 5],
 ['a1', 'b1', 6, 7]]
>>> arr.dump(axes_names='except_last')
[['a', 'b', 'c0', 'c1'],
 ['a0', 'b0', 0, 1],
 ['a0', 'b1', 2, 3],
 ['a1', 'b0', 4, 5],
 ['a1', 'b1', 6, 7]]
>>> arr.dump(light=True)
[['a', 'b\\c', 'c0', 'c1'],
 ['a0', 'b0', 0, 1],
 ['', 'b1', 2, 3],
 ['a1', 'b0', 4, 5],
 ['', 'b1', 6, 7]]
>>> arr.dump(wide=False, value_name='data')
[['a', 'b', 'c', 'data'],
 ['a0', 'b0', 'c0', 0],
 ['a0', 'b0', 'c1', 1],
 ['a0', 'b1', 'c0', 2],
 ['a0', 'b1', 'c1', 3],
 ['a1', 'b0', 'c0', 4],
 ['a1', 'b0', 'c1', 5],
 ['a1', 'b1', 'c0', 6],
 ['a1', 'b1', 'c1', 7]]
>>> arr.dump(maxlines=3, edgeitems=1)
[['a', 'b\\c', 'c0', 'c1'],
 ['a0', 'b0', 0, 1],
 ['...', '...', '...', '...'],
 ['a1', 'b1', 6, 7]]
```

4.3.9 Excel

`open_excel(filepath, overwrite_file, ...)`

Open an Excel workbook

larray.open_excel

`larray.open_excel` (*filepath=None, overwrite_file=False, visible=None, silent=None, app=None, load_addins=None*)

Open an Excel workbook

Parameters

filepath [None, int or str, optional] path to the Excel file. The file must exist if `overwrite_file` is False. Use None for a new blank workbook, -1 for the currently active workbook. Defaults to None.

overwrite_file [bool, optional] whether or not to overwrite an existing file, if any. Defaults to False.

visible [None or bool, optional] whether or not Excel should be visible. Defaults to False for files, True for new/active workbooks and to None (“unchanged”) for existing unsaved workbooks.

silent [None or bool, optional] whether or not to show dialog boxes for updating links or when some links cannot be updated. Defaults to False if visible, True otherwise.

app [None, “new”, “active”, “global” or `xlwings.App`, optional] use “new” for opening a new Excel instance, “active” for the last active instance (including ones opened by the user) and “global” to (re)use the same instance for all workbooks of a program. None is equivalent to “active” if `filepath` is -1, “new” if `visible` is True and “global” otherwise. Defaults to None.

The “global” instance is a specific Excel instance for all input from/output to Excel from within a single Python program (and should not interact with instances manually opened by the user or another program).

load_addins [None or bool, optional] whether or not to load Excel addins. Defaults to True if visible and `app == “new”`, False otherwise.

Returns

Excel workbook.

Examples

```
>>> arr = ndtest((3, 3))
>>> arr
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
a2    6   7   8
```

create a new Excel file and save an array

```
>>> # to create a new Excel file, argument overwrite_file must be set to True
>>> with open_excel('excel_file.xlsx', overwrite_file=True) as wb:
...     wb['arr'] = arr.dump()
...     wb.save()
```

read array from an Excel file


```

>>> with open_excel('excel_file.xlsx') as wb:
...     arr2 = wb['arr'].load()
>>> arr2
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
a2    6   7   8

```

<code>Workbook(filepath, overwrite_file, ...)</code>	Excel Workbook.
<code>Workbook.sheet_names(self)</code>	Returns the names of the Excel sheets.
<code>Workbook.save(self[, path])</code>	Saves the Workbook.
<code>Workbook.close(self)</code>	Close the workbook in Excel.
<code>Workbook.app(self)</code>	Return the Excel instance this workbook is attached to.

larray.Workbook

class `larray.Workbook` (*filepath=None, overwrite_file=False, visible=None, silent=None, app=None, load_addins=None*)

Excel Workbook.

See also:

`open_excel`

`__init__` (*self, filepath=None, overwrite_file=False, visible=None, silent=None, app=None, load_addins=None*)
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__(self[, filepath, overwrite_file, ...])</code>	Initialize self.
<code>app(self)</code>	Return the Excel instance this workbook is attached to.
<code>close(self)</code>	Close the workbook in Excel.
<code>save(self[, path])</code>	Saves the Workbook.
<code>sheet_names(self)</code>	Returns the names of the Excel sheets.

larray.Workbook.sheet_names

`Workbook.sheet_names` (*self*)
Returns the names of the Excel sheets.

Examples

```
>>> arr, arr2, arr3 = ndtest((3, 3)), ndtest((2, 2)), ndtest(4)
>>> with open_excel('excel_file.xlsx', overwrite_file=True) as wb:
...     wb['arr'] = arr.dump()
...     wb['arr2'] = arr2.dump()
...     wb['arr3'] = arr3.dump()
...     wb.save()
...
...     wb.sheet_names()
['arr', 'arr2', 'arr3']
```

larray.Workbook.save

Workbook.**save** (*self*, *path=None*)

Saves the Workbook.

If a path is being provided, this works like SaveAs() in Excel. If no path is specified and if the file hasn't been saved previously, it's being saved in the current working directory with the current filename. Existing files are overwritten without prompting.

Parameters

path [str, optional] Full path to the workbook. Defaults to None.

Examples

```
>>> arr, arr2, arr3 = ndtest((3, 3)), ndtest((2, 2)), ndtest(4)
>>> with open_excel('excel_file.xlsx', overwrite_file=True) as wb:
...     wb['arr'] = arr.dump()
...     wb['arr2'] = arr2.dump()
...     wb['arr3'] = arr3.dump()
...     wb.save()
```

larray.Workbook.close

Workbook.**close** (*self*)

Close the workbook in Excel.

Need to be called if the workbook has been opened without the *with* statement.

Examples

```
>>> arr, arr2, arr3 = ndtest((3, 3)), ndtest((2, 2)), ndtest(4)
>>> wb = open_excel('excel_file.xlsx', overwrite_file=True)
>>> wb['arr'] = arr.dump()
>>> wb['arr2'] = arr2.dump()
>>> wb['arr3'] = arr3.dump()
>>> wb.save()
>>> wb.close()
```

`larray.Workbook.app`

`Workbook.app(self)`

Return the Excel instance this workbook is attached to.

4.3.10 ExcelReport

<code>ExcelReport()</code>	Automate the generation of multiple graphs in an Excel file.
<code>ExcelReport.template_dir</code>	Set the path to the directory containing the Excel template files (with <code>.crtx</code> extension).
<code>ExcelReport.template</code>	Set a default Excel template file.
<code>ExcelReport.set_item_default_size(self, kind)</code>	Override the default <code>'width'</code> and <code>'height'</code> values for the given kind of item.
<code>ExcelReport.graphs_per_row</code>	Default number of graphs per row.
<code>ExcelReport.new_sheet(self, sheet_name)</code>	Add a new empty output sheet.
<code>ExcelReport.sheet_names(self)</code>	Returns the names of the output sheets.
<code>ExcelReport.to_excel(self, filepath[, ...])</code>	Generate the report Excel file.

`larray.ExcelReport`

class `larray.ExcelReport`

Automate the generation of multiple graphs in an Excel file.

The `ExcelReport` instance is initially populated with information (data, title, destination sheet, template, size) required to create the graphs. Once all information has been provided, the `to_excel` method is called to generate an Excel file with all graphs in one step.

Parameters

template_dir [str, optional] Path to the directory containing the Excel template files (with a `.crtx` extension). Defaults to `None`.

template [str, optional] Name of the template to be used as default template. The extension `.crtx` will be added if not given. The full path to the template file must be given if no template directory has been set. Defaults to `None`.

graphs_per_row: int, optional Default number of graphs per row. Defaults to 1.

Notes

The data associated with all graphical items is dumped in the same sheet named ‘__data__’.

Examples

```
>>> demo = load_example_data('demography_eurostat')
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
```

Set a new destination sheet

```
>>> sheet_be = report.new_sheet('Belgium')
```

Add a new title item

```
>>> sheet_be.add_title('Population, births and deaths')
```

Add a new graph item (each new graph is placed right to previous one unless you use `newline()` or `add_title()`)

```
>>> # using default 'width' and 'height' values
>>> sheet_be.add_graph(demo.pop['Belgium'], 'Population', template='Line')
>>> # specifying the 'width' and 'height' values
>>> sheet_be.add_graph(demo.births['Belgium'], 'Births', template='Line',
↳width=450, height=250)
```

Override the default ‘width’ and ‘height’ values for graphs

```
>>> sheet_be.set_item_default_size('graph', width=450, height=250)
>>> # add a new graph with the new default 'width' and 'height' values
>>> sheet_be.add_graph(demo.deaths['Belgium'], 'Deaths')
```

Set a default template for all next graphs

```
>>> # if a default template directory has been set, just pass the name
>>> sheet_be.template = 'Line'
>>> # otherwise, give the full path to the template file
>>> sheet_be.template = r'C:\other_template_dir\Line_Marker.crtx'
>>> # add a new graph with the default template
>>> sheet_be.add_graph(demo.pop['Belgium', 'Female'], 'Population - Female')
>>> sheet_be.add_graph(demo.pop['Belgium', 'Male'], 'Population - Male')
```

Specify the number of graphs per row

```
>>> sheet_countries = report.new_sheet('All countries')
```

```
>>> sheet_countries.graphs_per_row = 2
>>> for combined_labels, subset in demo.pop.items(('time', 'gender')):
...     title = '- '.join([str(label) for label in combined_labels])
...     sheet_countries.add_graph(subset, title)
```

Force a new row of graphs

```
>>> sheet_countries.newline()
```

Add multiple graphs at once (add a new graph for each combination of gender and year)

```
>>> sheet_countries.add_graphs({'Population of {gender} by country for the year
↳ {year}': pop},
...                             {'gender': pop.gender, 'year': pop.time},
...                             template='line', width=450, height=250, graphs_per_
↳ row=2)
```

Generate the report Excel file

```
>>> report.to_excel('Demography_Report.xlsx')
```

__init__(self)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self)</code>	Initialize self.
<code>new_sheet(self, sheet_name)</code>	Add a new empty output sheet.
<code>set_item_default_size(self, kind[, width, height[, ...]])</code>	Override the default ‘width’ and ‘height’ values for the given kind of item.
<code>sheet_names(self)</code>	Returns the names of the output sheets.
<code>to_excel(self, filepath[, data_sheet_name, ...])</code>	Generate the report Excel file.

Attributes

<code>graphs_per_row</code>	Default number of graphs per row.
<code>template</code>	Set a default Excel template file.
<code>template_dir</code>	Set the path to the directory containing the Excel template files (with ‘.ctx’ extension).

larray.ExcelReport.template_dir

property ExcelReport.**template_dir**

Set the path to the directory containing the Excel template files (with ‘.ctx’ extension). This method is mainly useful if your template files are located in several directories, otherwise pass the template directory directly the ExcelReport constructor.

Parameters

template_dir [str] Path to the directory containing the Excel template files.

See also:

set_graph_template

Examples

```
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
>>> # ... add some graphs using template files from 'C:\excel_templates_dir'
>>> report.template_dir = r'C:\other_templates_dir'
>>> # ... add some graphs using template files from 'C:\other_templates_dir'
```

larray.ExcelReport.template

property ExcelReport.template

Set a default Excel template file.

Parameters

template_file [str] Name of the template to be used as default template. The extension ‘.ctx’ will be added if not given. The full path to the template file must be given if no template directory has been set.

Examples

```
>>> demo = load_example_data('demography_eurostat')
```

Passing the name of the template (only if a template directory has been set)

```
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
>>> report.template = 'Line'
```

```
>>> sheet_pop = report.new_sheet('Population')
>>> sheet_pop.add_graph(demo.pop['Belgium'], 'Belgium')
```

Passing the full path of the template file

```
>>> # if no default template directory has been set
>>> # or if the new template is located in another directory,
>>> # you must provide the full path
>>> sheet_pop.template = r'C:\other_templates_dir\Line_Marker.ctx'
>>> sheet_pop.add_graph(demo.pop['Germany'], 'Germany')
```

larray.ExcelReport.set_item_default_size

ExcelReport.set_item_default_size(self, kind, width=None, height=None)

Override the default ‘width’ and ‘height’ values for the given kind of item. A new value must be provided at least for ‘width’ or ‘height’.

Parameters

kind [str] kind of item for which default values of ‘width’ and/or ‘height’ are modified. Currently available kinds are ‘title’ and ‘graph’.

width [int, optional] new default width value.

height [int, optional] new default height value.

Examples

```
>>> report = ExcelReport()
>>> report.set_item_default_size('graph', width=450, height=250)
```

larray.ExcelReport.graphs_per_row

property ExcelReport.graphs_per_row

Default number of graphs per row.

Parameters

graphs_per_row: int

See also:

ReportSheet.newline

larray.ExcelReport.new_sheet

ExcelReport.new_sheet(*self*, *sheet_name*)

Add a new empty output sheet. This sheet will contain only graphical elements, all data are exported to a dedicated separate sheet.

Parameters

sheet_name [str] name of the current sheet.

Returns

sheet: SheetReport

Examples

```
>>> demo = load_example_data('demography_eurostat')
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
```

```
>>> # prepare new output sheet named 'Belgium'
>>> sheet_be = report.new_sheet('Belgium')
```

```
>>> # add graph to the output sheet 'Belgium'
>>> sheet_be.add_graph(demo.pop['Belgium'], 'Population', template='Line')
```

larray.ExcelReport.sheet_names

ExcelReport.sheet_names(*self*)

Returns the names of the output sheets.

Examples

```
>>> report = ExcelReport()
>>> sheet_pop = report.new_sheet('Pop')
>>> sheet_births = report.new_sheet('Births')
>>> sheet_deaths = report.new_sheet('Deaths')
>>> report.sheet_names()
['Pop', 'Births', 'Deaths']
```

larray.ExcelReport.to_excel

`ExcelReport.to_excel(self, filepath, data_sheet_name='__data__', overwrite=True)`

Generate the report Excel file.

Parameters

filepath [str] Path of the report file for the dump.

data_sheet_name [str, optional] name of the Excel sheet where all data associated with items is dumped. Defaults to `'__data__'`.

overwrite [bool, optional] whether or not to overwrite an existing report file. Defaults to `True`.

Examples

```
>>> demo = load_example_data('demography_eurostat')
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
>>> report.template = 'Line_Marker'
```

```
>>> for c in demo.country:
...     sheet_country = report.new_sheet(c)
...     sheet_country.add_graph(demo.pop[c], 'Population')
...     sheet_country.add_graph(demo.births[c], 'Births')
...     sheet_country.add_graph(demo.deaths[c], 'Deaths')
```

Basic usage

```
>>> report.to_excel('Demography_Report.xlsx')
```

Alternative data sheet name

```
>>> report.to_excel('Demography_Report.xlsx', data_sheet_name='Data Tables')
```

Check if output file already exists

```
>>> report.to_excel('Demography_Report.xlsx', overwrite=False)
Traceback (most recent call last):
...
ValueError: Sheet named 'Belgium' already present in workbook
```


4.3.11 ReportSheet

<code>ReportSheet()</code>	Represents a sheet dedicated to contains only graphical items (title banners, graphs).
<code>ReportSheet.template_dir</code>	Set the path to the directory containing the Excel template files (with '.crtx' extension).
<code>ReportSheet.template</code>	Set a default Excel template file.
<code>ReportSheet.set_item_default_size(self, kind)</code>	Override the default 'width' and 'height' values for the given kind of item.
<code>ReportSheet.graphs_per_row</code>	Default number of graphs per row.
<code>ReportSheet.add_title(self, title[, width, ...])</code>	Add a title item to the current sheet.
<code>ReportSheet.add_graph(self, data[, title, ...])</code>	Add a graph item to the current sheet.
<code>ReportSheet.add_graphs(self, ...[, ...])</code>	Add multiple graph items to the current sheet.
<code>ReportSheet.newline(self)</code>	Force a new row of graphs.

larray.ReportSheet

class larray.ReportSheet

Represents a sheet dedicated to contains only graphical items (title banners, graphs). See [ExcelReport](#) for use cases.

Parameters

template_dir [str, optional] Path to the directory containing the Excel template files (with a '.crtx' extension). Defaults to None.

template [str, optional] Name of the template to be used as default template. The extension '.crtx' will be added if not given. The full path to the template file must be given if no template directory has been set. Defaults to None.

graphs_per_row: int, optional Default number of graphs per row. Defaults to 1.

See also:

[ExcelReport](#)

`__init__(self)`

Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__(self)</code>	Initialize self.
<code>add_graph(self, data[, title, template, ...])</code>	Add a graph item to the current sheet.
<code>add_graphs(self, array_per_title, ...[, ...])</code>	Add multiple graph items to the current sheet.
<code>add_title(self, title[, width, height, fontsize])</code>	Add a title item to the current sheet.
<code>newline(self)</code>	Force a new row of graphs.
<code>set_item_default_size(self, kind[, width, ...])</code>	Override the default 'width' and 'height' values for the given kind of item.

Attributes

<code>graphs_per_row</code>	Default number of graphs per row.
<code>template</code>	Set a default Excel template file.
<code>template_dir</code>	Set the path to the directory containing the Excel template files (with <code>‘.ctx’</code> extension).

`larray.ReportSheet.template_dir`

property `ReportSheet.template_dir`

Set the path to the directory containing the Excel template files (with `‘.ctx’` extension). This method is mainly useful if your template files are located in several directories, otherwise pass the template directory directly the `ExcelReport` constructor.

Parameters

template_dir [str] Path to the directory containing the Excel template files.

See also:

set_graph_template

Examples

```
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
>>> # ... add some graphs using template files from 'C:\excel_templates_dir'
>>> report.template_dir = r'C:\other_templates_dir'
>>> # ... add some graphs using template files from 'C:\other_templates_dir'
```

`larray.ReportSheet.template`

property `ReportSheet.template`

Set a default Excel template file.

Parameters

template_file [str] Name of the template to be used as default template. The extension `‘.ctx’` will be added if not given. The full path to the template file must be given if no template directory has been set.

Examples

```
>>> demo = load_example_data('demography_eurostat')
```

Passing the name of the template (only if a template directory has been set)

```
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
>>> report.template = 'Line'
```

```
>>> sheet_pop = report.new_sheet('Population')
>>> sheet_pop.add_graph(demo.pop['Belgium'], 'Belgium')
```

Passing the full path of the template file

```
>>> # if no default template directory has been set
>>> # or if the new template is located in another directory,
>>> # you must provide the full path
>>> sheet_pop.template = r'C:\other_templates_dir\Line_Marker.crtx'
>>> sheet_pop.add_graph(demo.pop['Germany'], 'Germany')
```

larray.ReportSheet.set_item_default_size

`ReportSheet.set_item_default_size(self, kind, width=None, height=None)`

Override the default ‘width’ and ‘height’ values for the given kind of item. A new value must be provided at least for ‘width’ or ‘height’.

Parameters

kind [str] kind of item for which default values of ‘width’ and/or ‘height’ are modified. Currently available kinds are ‘title’ and ‘graph’.

width [int, optional] new default width value.

height [int, optional] new default height value.

Examples

```
>>> report = ExcelReport()
>>> report.set_item_default_size('graph', width=450, height=250)
```

larray.ReportSheet.graphs_per_row

property `ReportSheet.graphs_per_row`

Default number of graphs per row.

Parameters

graphs_per_row: int

See also:

ReportSheet.newline

larray.ReportSheet.add_title

`ReportSheet.add_title(self, title, width=None, height=None, fontsize=11)`

Add a title item to the current sheet. Note that the current method only add a new item to the list of items to be generated. The report Excel file is generated only when the `to_excel` is called.

Parameters

title [str] Text to write in the title item.

width [int, optional] width of the title item. The current default value is used if None (see `set_item_default_size`). Defaults to None.

height [int, optional] height of the title item. The current default value is used if None (see `set_item_default_size`). Defaults to None.

fontsize [int, optional] fontsize of the displayed text. Defaults to 11.

Examples

```
>>> report = ExcelReport()
```

```
>>> first_sheet = report.new_sheet('First_sheet')
>>> first_sheet.add_title('Title banner with default width, height and fontsize')
>>> first_sheet.add_title('Larger title banner', width=1200, height=100)
>>> first_sheet.add_title('Bigger fontsize', fontsize=13)
```

```
>>> # do not forget to call 'to_excel' to create the report file
>>> report.to_excel('Report.xlsx')
```

larray.ReportSheet.add_graph

`ReportSheet.add_graph` (*self*, *data*, *title=None*, *template=None*, *width=None*, *height=None*)

Add a graph item to the current sheet. Note that the current method only add a new item to the list of items to be generated. The report Excel file is generated only when the `to_excel` is called.

Parameters

data [1D or 2D array-like] 1D or 2D array representing the data associated with the graph. The first row represents the abscissa labels. Each additional row represents a new series and must start with the name of the current series.

title [str, optional] title of the graph. Defaults to None.

template [str, optional] name of the template to be used to generate the graph. The full path to the template file must be provided if no template directory has not been set or if the template file belongs to another directory. Defaults to the defined template (see `set_graph_template`).

width [int, optional] width of the title item. The current default value is used if None (see `set_item_default_size`). Defaults to None.

height [int, optional] height of the title item. The current default value is used if None (see `set_item_default_size`). Defaults to None.

Examples

```
>>> demo = load_example_data('demography_eurostat')
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
```

```
>>> sheet_be = report.new_sheet('Belgium')
```

Specifying the ‘template’

```
>>> sheet_be.add_graph(demo.pop['Belgium'], 'Population', template='Line')
```

Specifying the ‘template’, ‘width’ and ‘height’ values

```
>>> sheet_be.add_graph(demo.births['Belgium'], 'Births', template='Line',
↳ width=450, height=250)
```

Setting a default template

```
>>> sheet_be.template = 'Line_Marker'
>>> sheet_be.add_graph(demo.deaths['Belgium'], 'Deaths')
```

Dumping the report Excel file

```
>>> # do not forget to call 'to_excel' to create the report file
>>> report.to_excel('Demography_Report.xlsx')
```

larray.ReportSheet.add_graphs

`ReportSheet.add_graphs` (*self*, *array_per_title*, *axis_per_loop_variable*, *template=None*, *width=None*, *height=None*, *graphs_per_row=1*)

Add multiple graph items to the current sheet. This method is mainly useful when multiple graphs are generated by iterating over one or several axes of an array (see examples below). The report Excel file is generated only when the `to_excel` is called.

Parameters

array_per_title: dict dictionary containing pairs (title template, array).

axis_per_loop_variable: dict dictionary containing pairs (variable used in the title template, axis).

template [str, optional] name of the template to be used to generate the graph. The full path to the template file must be provided if no template directory has not been set or if the template file belongs to another directory. Defaults to the defined template (see `set_graph_template`).

width [int, optional] width of the title item. The current default value is used if None (see `set_item_default_size`). Defaults to None.

height [int, optional] height of the title item. The current default value is used if None (see `set_item_default_size`). Defaults to None.

graphs_per_row: int, optional Number of graphs per row. Defaults to 1.

Examples

```
>>> demo = load_example_data('demography_eurostat')
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
```

```
>>> sheet_pop = report.new_sheet('Population')
>>> pop = demo.pop
```

Generate a new graph for each combination of gender and year

```
>>> sheet_pop.add_graphs({'Population of {gender} by country for the year {year}':
→ pop},
...                       {'gender': pop.gender, 'year': pop.time},
...                       template='line', width=450, height=250, graphs_per_row=2)
```

```
>>> # do not forget to call 'to_excel' to create the report file
>>> report.to_excel('Demography_Report.xlsx')
```

larray.ReportSheet.newline

`ReportSheet.newline` (*self*)
Force a new row of graphs.

4.3.12 Miscellaneous

<code>asarray(a[, meta])</code>	Converts input as Array if possible.
<code>from_frame(df[, sort_rows, sort_columns, ...])</code>	Converts Pandas DataFrame into Array.
<code>from_series(s[, sort_rows, fill_value, meta])</code>	Converts Pandas Series into Array.
<code>get_example_filepath(fname)</code>	Return absolute path to an example file if exist.
<code>set_options(**kwargs)</code>	Set options for larray in a controlled context.
<code>get_options()</code>	Return the current options.
<code>labels_array(axes[, title, meta])</code>	Returns an array with specified axes and the combination of corresponding labels as values.
<code>union(*args)</code>	Returns the union of several “value strings” as a list.
<code>stack([elements, axes, title, meta, dtype, ...])</code>	Combines several arrays or sessions along an axis.
<code>identity(axis)</code>	
<code>diag(a[, k, axes, ndim, split])</code>	Extracts a diagonal or construct a diagonal array.
<code>eye(rows[, columns, k, title, dtype, meta])</code>	Returns a 2-D array with ones on the diagonal and zeros elsewhere.
<code>ipfp(target_sums[, a, axes, maxiter, ...])</code>	Apply Iterative Proportional Fitting Procedure (also known as bi-proportional fitting in statistics, RAS algorithm in economics) to array a, with target_sums as targets.
<code>wrap_elementwise_array_func(func)</code>	Wrap a function using numpy arrays to work with LArray arrays instead.
<code>zip_array_values(values[, axes, ascending])</code>	Returns a sequence as if simultaneously iterating on several arrays.
<code>zip_array_items(values[, axes, ascending])</code>	Returns a sequence as if simultaneously iterating on several arrays as well as the current iteration “key”.

larray.asarray

`larray.asarray` (*a, meta=None*)
Converts input as Array if possible.

Parameters

a [array-like] Input array to convert into an Array.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> # NumPy array
>>> np_arr = np.arange(6).reshape((2,3))
>>> asarray(np_arr)
{0}\{1}*  0  1  2
          0  0  1  2
          1  3  4  5
>>> # Pandas dataframe
>>> data = {'normal' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
...        'reverse' : pd.Series([3., 2., 1.], index=['a', 'b', 'c'])}
>>> df = pd.DataFrame(data)
>>> asarray(df)
{0}\{1}  normal  reverse
a         1.0     3.0
b         2.0     2.0
c         3.0     1.0
```

larray.from_frame

`larray.from_frame(df, sort_rows=False, sort_columns=False, parse_header=False, unfold_last_axis_name=False, fill_value=nan, meta=None, cartesian_prod=True, **kwargs)`

Converts Pandas DataFrame into Array.

Parameters

df [pandas.DataFrame] Input dataframe. By default, name and labels of the last axis are defined by the name and labels of the columns Index of the dataframe unless argument `unfold_last_axis_name` is set to True.

sort_rows [bool, optional] Whether or not to sort the rows alphabetically (sorting is more efficient than not sorting). Must be False if `cartesian_prod` is set to True. Defaults to False.

sort_columns [bool, optional] Whether or not to sort the columns alphabetically (sorting is more efficient than not sorting). Must be False if `cartesian_prod` is set to True. Defaults to False.

parse_header [bool, optional] Whether or not to parse columns labels. Pandas treats column labels as strings. If True, column labels are converted into int, float or boolean when possible. Defaults to False.

unfold_last_axis_name [bool, optional] Whether or not to extract the names of the last two axes by splitting the name of the last index column of the dataframe using `\`. Defaults to False.

fill_value [scalar, optional] Value used to fill cells corresponding to label combinations which are not present in the input DataFrame. Defaults to NaN.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

cartesian_prod [bool, optional] Whether or not to expand the dataframe to a cartesian product dataframe as needed by Array. This is an expensive operation but is absolutely required if you cannot guarantee your dataframe is already well formed. If True, arguments `sort_rows` and `sort_columns` must be set to False. Defaults to True.

Returns**Array****See also:**[*Array.to_frame*](#)**Examples**

```
>>> from larray import ndtest
>>> df = ndtest((2, 2, 2)).to_frame()
>>> df
c      c0  c1
a  b
a0 b0    0   1
    b1    2   3
a1 b0    4   5
    b1    6   7
>>> from_frame(df)
a  b\c  c0  c1
a0 b0    0   1
a0 b1    2   3
a1 b0    4   5
a1 b1    6   7
```

Names of the last two axes written as before_last_axis_name\\last_axis_name

```
>>> df = ndtest((2, 2, 2)).to_frame(fold_last_axis_name=True)
>>> df
      c0  c1
a  b\c
a0 b0    0   1
    b1    2   3
a1 b0    4   5
    b1    6   7
>>> from_frame(df, unfold_last_axis_name=True)
a  b\c  c0  c1
a0 b0    0   1
a0 b1    2   3
a1 b0    4   5
a1 b1    6   7
```

larray.from_series**larray.from_series** (*s*, *sort_rows=False*, *fill_value=nan*, *meta=None*, ***kwargs*)

Converts Pandas Series into Array.

Parameters**s** [Pandas Series] Input Pandas Series.**sort_rows** [bool, optional] Whether or not to sort the rows alphabetically. Defaults to False.**fill_value** [scalar, optional] Value used to fill cells corresponding to label combinations which are not present in the input Series. Defaults to NaN.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

See also:

[*Array.to_series*](#)

Examples

```
>>> from larray import ndtest
>>> s = ndtest((2, 2, 2), dtype=float).to_series()
>>> s
a   b   c
a0  b0  c0    0.0
      c1    1.0
      b1  c0    2.0
      c1    3.0
a1  b0  c0    4.0
      c1    5.0
      b1  c0    6.0
      c1    7.0
dtype: float64
>>> from_series(s)
a  b\c   c0   c1
a0  b0  0.0  1.0
a0  b1  2.0  3.0
a1  b0  4.0  5.0
a1  b1  6.0  7.0
```

`larray.get_example_filepath`

`larray.get_example_filepath(fname)`

Return absolute path to an example file if exist.

Parameters

fname [str] Filename of an existing example file.

Returns

Filepath Absolute filepath to an example file if exists.

Notes

A `ValueError` is raised if the provided filename does not represent an existing example file.

Examples

```
>>> fpath = get_example_filepath('examples.xlsx')
```

`larray.set_options`

class `larray.set_options` (***kwargs*)

Set options for `larray` in a controlled context.

Currently supported options:

- `display_precision`: number of digits of precision for floating point output. Print as many digits as necessary to uniquely specify the value by default (None).
- `display_width`: maximum display width for `repr` on `larray` objects. Defaults to 80.
- `display_maxlines`: Maximum number of lines to show. All lines are shown if -1. Defaults to 200.
- `display_edgeitems`: if number of lines to display is greater than `display_maxlines`, only the first and last `display_edgeitems` lines are displayed. Only active if `display_maxlines` is not -1. Defaults to 5.

Examples

```
>>> from larray import *
>>> arr = ndtest((500, 100), dtype=float) + 0.123456
```

You can use `set_options` either as a context manager:

```
>>> with set_options(display_width=100, display_edgeitems=2):
...     print(arr)
a\b      b0      b1      b2 ...      b97      b98
↪      b99
a0      0.123456      1.123456      2.123456 ...      97.123456      98.123456
↪ 99.123456
a1      100.123456      101.123456      102.123456 ...      197.123456      198.123456
↪ 199.123456
...      ...      ...      ...      ...      ...
↪      ...
a498  49800.123456  49801.123456  49802.123456 ...  49897.123456  49898.123456
↪ 49899.123456
a499  49900.123456  49901.123456  49902.123456 ...  49997.123456  49998.123456
↪ 49999.123456
```

Or to set global options:

```
>>> set_options(display_maxlines=10, display_precision=2)
>>> print(arr)
a\b      b0      b1      b2 ...      b97      b98      b99
a0      0.12      1.12      2.12 ...      97.12      98.12      99.12
a1      100.12     101.12     102.12 ...      197.12     198.12     199.12
```

(continues on next page)

(continued from previous page)

a2	200.12	201.12	202.12	...	297.12	298.12	299.12
a3	300.12	301.12	302.12	...	397.12	398.12	399.12
a4	400.12	401.12	402.12	...	497.12	498.12	499.12
...
a495	49500.12	49501.12	49502.12	...	49597.12	49598.12	49599.12
a496	49600.12	49601.12	49602.12	...	49697.12	49698.12	49699.12
a497	49700.12	49701.12	49702.12	...	49797.12	49798.12	49799.12
a498	49800.12	49801.12	49802.12	...	49897.12	49898.12	49899.12
a499	49900.12	49901.12	49902.12	...	49997.12	49998.12	49999.12

To put back the default options, you can use:

```
>>> set_options(display_precision=None, display_width=80, display_maxlines=200,
↳display_edgeitems=5)
...
```

`__init__(self, **kwargs)`

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__(self, **kwargs)`

Initialize self.

`larray.get_options`

`larray.get_options()`

Return the current options.

Returns

Dictionary of current print options with keys

- `display_precision`: int or None
- `display_width`: int
- `display_maxlines`: int
- `display_edgeitems`: int

For a full description of these options, see [set_options](#).

See also:

[set_options](#)

Examples

```
>>> get_options()
{'display_precision': None, 'display_width': 80, 'display_maxlines': 200,
 ↪ 'display_edgeitems': 5}
```

larray.labels_array

`larray.labels_array` (*axes*, *title=None*, *meta=None*)

Returns an array with specified axes and the combination of corresponding labels as values.

Parameters

axes [Axis or collection of Axis]

title [str, optional] Deprecated. See ‘meta’ below.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> labels_array(sex)
sex  M  F
     M  F
>>> labels_array((nat, sex))
nat  sex\axis  nat  sex
BE      M    BE   M
BE      F    BE   F
FO      M    FO   M
FO      F    FO   F
```

larray.union

`larray.union` (**args*)

Returns the union of several “value strings” as a list.

Parameters

***args** (collection of) value(s) to be converted into label(s). Repeated values are taken only once.

Returns

list of labels

Examples

```
>>> union('a', 'a, b, c, d', ['d', 'e', 'f'], '..2')
['a', 'b', 'c', 'd', 'e', 'f', 0, 1, 2]
```

larray.stack

`larray.stack`(*elements=None*, *axes=None*, *title=None*, *meta=None*, *dtype=None*, *res_axes=None*, ***kwargs*)

Combines several arrays or sessions along an axis.

Parameters

elements [tuple, list or dict.] Elements to stack. Elements can be scalars, arrays, sessions, (label, value) pairs or a {label: value} mapping. In the later case, axis must be defined and cannot be a name only, because we need to have labels order, which the mapping does not provide.

Stacking sessions will return a new session containing the arrays of all sessions stacked together. An array missing in a session will be replaced by NaN.

axes [str, Axis, Group or sequence of Axis, optional] Axes to create. If None, defaults to a `range()` axis.

title [str, optional] Deprecated. See ‘meta’ below.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

dtype [type, optional] Output dtype. Defaults to None (inspect all output values to infer it automatically).

res_axes [AxisCollection, optional] Axes of the output. Defaults to None (union of axes of all values and the stacking axes).

Returns

Array A single array combining arrays. The new (stacked) axes will be the last axes of the new array.

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> arr1 = ones(sex)
>>> arr1
sex    M    F
      1.0  1.0
>>> arr2 = zeros(sex)
>>> arr2
sex    M    F
      0.0  0.0
```

In case the axis to create has already been defined in a variable (Axis or Group)

```
>>> stack({'BE': arr1, 'FO': arr2}, nat)
sex\nat    BE    FO
      M  1.0  0.0
      F  1.0  0.0
```

Otherwise (when one wants to create an axis from scratch), any of these syntaxes works:

```
>>> stack([arr1, arr2], 'nat=BE,FO')
sex\nat    BE    FO
      M  1.0  0.0
      F  1.0  0.0
>>> stack({'BE': arr1, 'FO': arr2}, 'nat=BE,FO')
sex\nat    BE    FO
      M  1.0  0.0
      F  1.0  0.0
>>> stack([('BE', arr1), ('FO', arr2)], 'nat=BE,FO')
sex\nat    BE    FO
      M  1.0  0.0
      F  1.0  0.0
```

When stacking arrays with different axes, the result has the union of all axes present:

```
>>> stack({'BE': arr1, 'FO': 0}, nat)
sex\nat    BE    FO
      M  1.0  0.0
      F  1.0  0.0
```

Creating an axis without name nor labels can be done using:

```
>>> stack((arr1, arr2))
sex\{1}*    0    1
      M  1.0  0.0
      F  1.0  0.0
```

When labels are “simple” strings (ie no integers, no string starting with integers, etc.), using keyword arguments can be an attractive alternative.

```
>>> stack(FO=arr2, BE=arr1, axes=nat)
sex\nat    BE    FO
      M  1.0  0.0
      F  1.0  0.0
```

Without passing an explicit order for labels (or an axis object like above), it should only be used on Python 3.6 or later because keyword arguments are NOT ordered on earlier Python versions.

```
>>> # use this only on Python 3.6 and later
>>> stack(BE=arr1, FO=arr2, axes='nat')
sex\nat    BE    FO
      M  1.0  0.0
      F  1.0  0.0
```

One can also stack along several axes

```
>>> test = Axis('test=T1,T2')
>>> stack({'BE', 'T1': arr1,
...      ('BE', 'T2': arr2,
...      ('FO', 'T1': arr2,
```

(continues on next page)

(continued from previous page)

```

...      ('FO', 'T2'): arr1},
...      (nat, test))
sex  nat\test  T1  T2
M      BE  1.0  0.0
M      FO  0.0  1.0
F      BE  1.0  0.0
F      FO  0.0  1.0

```

To stack sessions, let us first create two test sessions. For example suppose we have a session storing the results of a baseline simulation:

```

>>> from larray import Session
>>> baseline = Session([('arr1', arr1), ('arr2', arr2)])

```

and another session with a variant (here we simply added 0.5 to each array)

```

>>> variant = Session([('arr1', arr1 + 0.5), ('arr2', arr2 + 0.5)])

```

then we stack them together

```

>>> stacked = stack([('baseline', baseline), ('variant', variant)], 'sessions')
>>> stacked
Session(arr1, arr2)
>>> stacked.arr1
sex\sessions  baseline  variant
M              1.0      1.5
F              1.0      1.5
>>> stacked.arr2
sex\sessions  baseline  variant
M              0.0      0.5
F              0.0      0.5

```

larray.identity

`larray.identity` (*axis*)

larray.diag

`larray.diag` (*a*, *k=0*, *axes=(0, 1)*, *ndim=2*, *split=True*)

Extracts a diagonal or construct a diagonal array.

Parameters

- a** [Array] If *a* has 2 dimensions or more, return a copy of its *k*-th diagonal. If *a* has 1 dimension, return an array with *ndim* dimensions on the *k*-th diagonal.
- k** [int, optional] Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).
- axes** [tuple or list or AxisCollection of axes references, optional] Axes along which the diagonals should be taken. Use None for all axes. Defaults to the first two axes (0, 1).
- ndim** [int, optional] Target number of dimensions when constructing a diagonal array from an array without axes names/labels. Defaults to 2.
- split** [bool, optional] Whether or not to try to split the axis name and labels. Defaults to True.

Returns

Array The extracted diagonal or constructed diagonal array.

Examples

```
>>> nat = Axis('nat=BE,FO')
>>> sex = Axis('sex=M,F')
>>> a = ndtest([nat, sex], start=1)
>>> a
nat\sex  M  F
      BE  1  2
      FO  3  4
>>> d = diag(a)
>>> d
nat_sex  BE_M  FO_F
          1     4
>>> diag(d)
nat\sex  M  F
      BE  1  0
      FO  0  4
>>> a = ndtest(sex, start=1)
>>> a
sex  M  F
     1  2
>>> diag(a)
sex\sex  M  F
      M  1  0
      F  0  2
```

larray.eye

`larray.eye (rows, columns=None, k=0, title=None, dtype=None, meta=None)`

Returns a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters

rows [int or Axis] Rows of the output.

columns [int or Axis, optional] Columns of the output. If None, defaults to rows.

k [int, optional] Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

title [str, optional] Deprecated. See ‘meta’ below.

dtype [data-type, optional] Data-type of the returned array. Defaults to float.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array of shape (rows, columns) An array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

Examples

```
>>> eye(2, dtype=int)
{0}*\\{1}*   0   1
              0   1   0
              1   0   1

>>> sex = Axis('sex=M,F')
>>> eye(sex)
sex\\sex      M      F
      M   1.0   0.0
      F   0.0   1.0

>>> age = Axis('age=0..2')
>>> eye(age, sex)
age\\sex      M      F
      0   1.0   0.0
      1   0.0   1.0
      2   0.0   0.0

>>> eye(3, k=1)
{0}*\\{1}*   0      1      2
              0  0.0   1.0   0.0
              1  0.0   0.0   1.0
              2  0.0   0.0   0.0
```

larray.ipfp

`larray.ipfp(target_sums, a=None, axes=None, maxiter=1000, threshold=0.5, stepstoabort=10, nzvzs='raise', no_convergence='raise', display_progress=False)`

Apply Iterative Proportional Fitting Procedure (also known as bi-proportional fitting in statistics, RAS algorithm in economics) to array `a`, with `target_sums` as targets.

Parameters

target_sums [tuple/list of array-like] Target sums to achieve. First element must be the sum to achieve along axis 0, the second the sum along axis 1, ...

a [array-like, optional] Starting values to fit, if not given starts with an array filled with 1.

axes [list/tuple of axes, optional] Axes on which the fitting procedure should be applied. Defaults to all axes.

maxiter [int, optional] Maximum number of iteration, defaults to 1000.

threshold [float, optional] Threshold below which the result is deemed acceptable, defaults to 0.5.

stepstoabort [int, optional] Number of consecutive steps with no improvement after which to abort. Defaults to 10.

nzvzs ['fix', 'warn' or 'raise', optional] Behavior when detecting non zero values where the sum is zero 'fix': set to zero (silently) 'warn': set to zero and print a warning 'raise': raise an exception (default)

no_convergence ['ignore', 'warn' or 'raise, optional] Behavior when the algorithm does not seem to converge. This condition is triggered both when the maximum number of iteration is reached or when the maximum absolute difference between the target and the current sums does not improve for `stepstoabort` iterations. 'ignore': return values computed up to that point (silently) 'warn': return values computed up to that point and print a warning 'raise': raise an exception (default)

display_progress [False, True or 'condensed', optional] Whether or not to display progress. Defaults to False. If 'condensed' will display progress using a denser template (using one line per iteration).

Returns

Array

Examples

```
>>> from larray import *
>>> a = Axis('a=a0,a1')
>>> b = Axis('b=b0,b1')
>>> initial = Array([[2, 1], [1, 2]], [a, b])
>>> initial
a\b  b0  b1
a0   2   1
a1   1   2
>>> target_sum_along_a = Array([2, 1], b)
>>> target_sum_along_a
b  b0  b1
   2   1
>>> target_sum_along_b = Array([1, 2], a)
>>> target_sum_along_b
a  a0  a1
   1   2
>>> result = ipfp([target_sum_along_a, target_sum_along_b], initial, threshold=0.
↳01)
>>> # round result so that its display is nicer
... round(result, 2)
a\b  b0  b1
a0   0.85  0.15
a1   1.15  0.85
```

Now let us assume you have a 3D array like this:

```
>>> year = Axis('year=2014..2016')
>>> initial = ndtest([a, b, year])
>>> initial
a  b\year  2014  2015  2016
a0  b0      0      1      2
a0  b1      3      4      5
a1  b0      6      7      8
a1  b1      9     10     11
```

and some targets for each year:

```
>>> btargets = initial.sum(X.a) + 1
>>> btargets
b\year  2014  2015  2016
b0      7      9     11
b1     13     15     17
>>> atargets = initial.sum(X.b) + 1
>>> atargets
a\year  2014  2015  2016
a0      4      6      8
a1     16     18     20
```

You want to apply a 2D fitting procedure for each value of that year axis. You could call `ipfp` within a loop on the year axis, but you can also apply the procedure for all years at once by using the `axes` argument. This is *much* faster than an explicit loop.

```
>>> result = ipfp([btargets, atargets], initial, axes=(X.a, X.b))
```

`larray.wrap_elementwise_array_func`

`larray.wrap_elementwise_array_func(func)`

Wrap a function using numpy arrays to work with LArray arrays instead.

Parameters

func [function] A function taking numpy arrays as arguments and returning numpy arrays of the same shape. If the function takes several arguments, this wrapping code assumes the result will have the combination of all axes present. In numpy talk, arguments will be broadcasted to each other.

Returns

function A function taking LArray arrays arguments and returning LArray arrays.

Examples

For example, if we want to apply the Hodrick-Prescott filter from `statsmodels` we can use this:

```
>>> from statsmodels.tsa.filters.hp_filter import hpfilter
>>> hpfilter = wrap_elementwise_array_func(hpfilter)
```

`hpfilter` is now a function taking a one dimensional Array as input and returning a one dimensional Array as output

Now let us suppose we have a ND array such as:

```
>>> from larray.random import normal
>>> arr = normal(axes="sex=M,F;year=2016..2018")
>>> arr
sex\year    2016    2017    2018
      M   -1.15    0.56   -1.06
      F   -0.48   -0.39   -0.98
```

We can apply an Hodrick-Prescott filter to it by using:

```
>>> # 6.25 is the recommended smoothing value for annual data
>>> cycle, trend = arr.apply(hpfilter, 6.25, axes="year")
>>> trend
sex\year    2016    2017    2018
      M   -0.61   -0.52   -0.52
      F   -0.37   -0.61   -0.87
```

larray.zip_array_values

`larray.zip_array_values` (*values*, *axes=None*, *ascending=True*)

Returns a sequence as if simultaneously iterating on several arrays.

Parameters

axes [int, str or Axis or tuple of them, optional] Axis or axes along which to iterate and in which order. Defaults to None (union of all axes present in all arrays, in the order they are found).

ascending [bool, optional] Whether or not to iterate the axes in ascending order (from start to end). Defaults to True.

Returns

Sequence

Examples

```
>>> arr1 = ndtest('a=a0,a1;b=b1,b2')
>>> arr2 = ndtest('a=a0,a1;c=c1,c2')
>>> arr1
a\b  b1  b2
a0   0   1
a1   2   3
>>> arr2
a\c  c1  c2
a0   0   1
a1   2   3
>>> for a1, a2 in zip_array_values((arr1, arr2), 'a'):
...     print("==")
...     print(a1)
...     print(a2)
==
b  b1  b2
   0   1
c  c1  c2
   0   1
==
b  b1  b2
   2   3
c  c1  c2
   2   3
>>> for a1, a2 in zip_array_values((arr1, arr2), arr2.c):
...     print("==")
...     print(a1)
...     print(a2)
==
a\b  b1  b2
a0   0   1
a1   2   3
a  a0  a1
   0   2
==
a\b  b1  b2
a0   0   1
a1   2   3
```

(continues on next page)

(continued from previous page)

```

a  a0  a1
   1   3
>>> for a1, a2 in zip_array_values((arr1, arr2)):
...     print("arr1: {}, arr2: {}".format(a1, a2))
arr1: 0, arr2: 0
arr1: 0, arr2: 1
arr1: 1, arr2: 0
arr1: 1, arr2: 1
arr1: 2, arr2: 2
arr1: 2, arr2: 3
arr1: 3, arr2: 2
arr1: 3, arr2: 3

```

larray.zip_array_items

`larray.zip_array_items` (*values*, *axes=None*, *ascending=True*)

Returns a sequence as if simultaneously iterating on several arrays as well as the current iteration “key”.

Broadcasts all values against each other. Scalars are simply repeated.

Parameters

values [Iterable] arrays to iterate on.

axes [int, str or Axis or tuple of them, optional] Axis or axes along which to iterate and in which order. Defaults to None (union of all axes present in all arrays, in the order they are found).

ascending [bool, optional] Whether or not to iterate the axes in ascending order (from start to end). Defaults to True.

Returns

Sequence

Examples

```

>>> arr1 = ndtest('a=a0,a1;b=b0,b1')
>>> arr2 = ndtest('a=a0,a1;c=c0,c1')
>>> arr1
a\b  b0  b1
a0   0   1
a1   2   3
>>> arr2
a\c  c0  c1
a0   0   1
a1   2   3
>>> for k, (a1, a2) in zip_array_items((arr1, arr2), 'a'):
...     print("==" , k[0], "==")
...     print(a1)
...     print(a2)
== a0 ==
b  b0  b1
   0   1
c  c0  c1
   0   1

```

(continues on next page)

(continued from previous page)

```

== a1 ==
b  b0  b1
   2   3
c  c0  c1
   2   3
>>> for k, (a1, a2) in zip_array_items((arr1, arr2), arr2.c):
...     print("==", k[0], "==")
...     print(a1)
...     print(a2)
== c0 ==
a\b  b0  b1
a0   0   1
a1   2   3
a  a0  a1
    0   2
== c1 ==
a\b  b0  b1
a0   0   1
a1   2   3
a  a0  a1
    1   3
>>> for k, (a1, a2) in zip_array_items((arr1, arr2)):
...     print(k, "arr1: {}, arr2: {}".format(a1, a2))
(a.i[0], b.i[0], c.i[0]) arr1: 0, arr2: 0
(a.i[0], b.i[0], c.i[1]) arr1: 0, arr2: 1
(a.i[0], b.i[1], c.i[0]) arr1: 1, arr2: 0
(a.i[0], b.i[1], c.i[1]) arr1: 1, arr2: 1
(a.i[1], b.i[0], c.i[0]) arr1: 2, arr2: 2
(a.i[1], b.i[0], c.i[1]) arr1: 2, arr2: 3
(a.i[1], b.i[1], c.i[0]) arr1: 3, arr2: 2
(a.i[1], b.i[1], c.i[1]) arr1: 3, arr2: 3

```

4.3.13 Session

<code>Session(*args, **kwargs)</code>	Groups several objects together.
<code>arrays([depth, include_private, meta])</code>	Returns a session containing all available arrays (whether they are defined in local or global variables) sorted in alphabetical order.
<code>local_arrays([depth, include_private, meta])</code>	Returns a session containing all local arrays sorted in alphabetical order.
<code>global_arrays([depth, include_private, meta])</code>	Returns a session containing all global arrays sorted in alphabetical order.
<code>load_example_data(name)</code>	Load arrays used in the tutorial so that all examples in it can be reproduced.

larray.Session

class larray.Session(*args, **kwargs)

Groups several objects together.

Parameters

***args** [str or dict of {str: object} or iterable of tuples (str, object)] Path to the file containing the session to load or list/tuple/dictionary containing couples (name, object).

****kwargs** [dict of {str: object}]

- **Objects** to add written as name=object
- **meta** [list of pairs or dict or OrderedDict or Metadata] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Warning: Metadata is not kept when actions or methods are applied on a session except for operations modifying a specific array, such as: `s['arr1'] = 0`. Do not add metadata to a session if you know you will apply actions or methods on it before dumping it.

Examples

```
>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
```

create a Session by passing a list of pairs (name, object)

```
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
```

create a Session using keyword arguments (but you lose order on Python < 3.6)

```
>>> s = Session(a=a, b=b, a01=a01, arr1=arr1, arr2=arr2)
```

create a Session by passing a dictionary (but you lose order on Python < 3.6)

```
>>> s = Session({'a': a, 'b': b, 'a01': a01, 'arr1': arr1, 'arr2': arr2})
```

load Session from file

```
>>> s = Session('my_session.h5')
```

create a session with metadata

```
>>> # Python <= 3.5
>>> s = Session([('arr1', arr1), ('arr2', arr2)], meta=[('title', 'my title'), (
↪'author', 'John Smith')])
>>> s.meta
title: my title
```

(continues on next page)

(continued from previous page)

```

author: John Smith
>>> # Python 3.6+
>>> s = Session(arr1=arr1, arr2=arr2, meta=Metadata(title='my title', author=
↪ 'John Smith'))
>>> s.meta
title: my title
author: John Smith

```

`__init__(self, *args, **kwargs)`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(self, *args, **kwargs)</code>	Initialize self.
<code>add(self, *args, **kwargs)</code>	Adds objects to the current session.
<code>apply(self, func, *args, **kwargs)</code>	Apply function <i>func</i> on elements of the session and return a new session.
<code>array_equals(*args, **kwargs)</code>	
<code>compact(self[, display])</code>	Detects and removes “useless” axes (ie axes for which values are constant over the whole axis) for all array objects in session
<code>copy(self)</code>	Returns a copy of the session.
<code>dump(*args, **kwargs)</code>	
<code>dump_csv(*args, **kwargs)</code>	
<code>dump_excel(*args, **kwargs)</code>	
<code>dump_hdf(*args, **kwargs)</code>	
<code>element_equals(self, other)</code>	Test if each element (group, axis and array) of the current session equals the corresponding element of another session.
<code>equals(self, other)</code>	Test if all elements (groups, axes and arrays) of the current session are equal to those of another session.
<code>filter(self[, pattern, kind])</code>	Returns a new session with objects which match some criteria.
<code>get(self, key[, default])</code>	Returns the object corresponding to the key.
<code>items(self)</code>	Returns a view of the session’s items ((key, value) pairs).
<code>keys(self)</code>	Returns a view on the session’s keys.
<code>load(self, fname[, names, engine, display])</code>	Load Array objects from a file, or several .csv files (all formats).
<code>save(self, fname[, names, engine, ...])</code>	Dumps Array objects from the current session to a file (all formats).
<code>summary(self[, template])</code>	Returns a summary of the content of the session.
<code>to_csv(self, fname[, names, display])</code>	Dumps Array objects from the current session to CSV files.
<code>to_excel(self, fname[, names, overwrite, ...])</code>	Dumps Array objects from the current session to an Excel file.
<code>to_globals(self[, names, depth, warn, inplace])</code>	Create global variables out of objects in the session.
<code>to_hdf(self, fname[, names, overwrite, display])</code>	Dumps Array, Axis and Group objects from the current session to an HDF file.

Continued on next page

Table 66 – continued from previous page

<code>to_pickle(self, fname[, names, overwrite, ...])</code>	Dumps Array, Axis and Group objects from the current session to a file using pickle.
<code>transpose(self, *args)</code>	Reorder axes of arrays in session, ignoring missing axes for each array.
<code>update(self[, other])</code>	Update the session with the key/value pairs from other or passed keyword arguments, overwriting existing keys.
<code>values(self)</code>	Returns a view on the session's values.

Attributes

<code>names</code>	Returns the list of names of the objects in the session.
--------------------	--

larray.arrays

`larray.arrays(depth=0, include_private=False, meta=None)`

Returns a session containing all available arrays (whether they are defined in local or global variables) sorted in alphabetical order. Local arrays take precedence over global ones (if a name corresponds to both a local and a global variable, the local array will be returned).

Parameters

depth: int depth of call frame to inspect. 0 is where *arrays* was called, 1 the caller of *arrays*, etc.

include_private: boolean, optional Whether or not to include private arrays (i.e. arrays starting with `_`). Defaults to False.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Session

larray.local_arrays

`larray.local_arrays(depth=0, include_private=False, meta=None)`

Returns a session containing all local arrays sorted in alphabetical order.

Parameters

depth: int depth of call frame to inspect. 0 is where *local_arrays* was called, 1 the caller of *local_arrays*, etc.

include_private: boolean, optional Whether or not to include private local arrays (i.e. arrays starting with `_`). Defaults to False.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Session

`larray.global_arrays`

`larray.global_arrays` (*depth=0, include_private=False, meta=None*)

Returns a session containing all global arrays sorted in alphabetical order.

Parameters

depth: **int** depth of call frame to inspect. 0 is where *global_arrays* was called, 1 the caller of *global_arrays*, etc.

include_private: **boolean, optional** Whether or not to include private globals arrays (i.e. arrays starting with `_`). Defaults to False.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Session

`larray.load_example_data`

`larray.load_example_data` (*name*)

Load arrays used in the tutorial so that all examples in it can be reproduced.

Parameters

name [str] Example data to load. Available example datasets are:

- demography
- demography_eurostat

Returns

Session Session containing one or several arrays.

Examples

```
>>> demo = load_example_data('demography')
>>> print(demo.summary())
hh: time, geo, hh_type (26 x 3 x 7) [int64]
pop: time, geo, age, sex, nat (26 x 3 x 121 x 2 x 2) [int64]
qx: time, geo, age, sex, nat (26 x 3 x 121 x 2 x 2) [float64]
>>> demo = load_example_data('demography_eurostat')
>>> print(demo.summary())
Metadata:
  title: Demographic datasets for a small selection of countries in Europe
  source: demo_jpan, demo_fasec, demo_magec and migr_imm1ctz tables from Eurostat
gender: gender ['Male' 'Female'] (2)
country: country ['Belgium' 'France' 'Germany'] (3)
country_benelux: country_benelux ['Belgium' 'Luxembourg' 'Netherlands'] (3)
citizenship: citizenship ['Belgium' 'Luxembourg' 'Netherlands'] (3)
time: time [2013 2014 2015 2016 2017] (5)
even_years: time[2014 2016] >> even_years (2)
```

(continues on next page)

(continued from previous page)

```

odd_years: time[2013 2015 2017] >> odd_years (3)
births: country, gender, time (3 x 2 x 5) [int32]
deaths: country, gender, time (3 x 2 x 5) [int32]
immigration: country, citizenship, gender, time (3 x 3 x 2 x 5) [int32]
pop: country, gender, time (3 x 2 x 5) [int32]
pop_benelux: country, gender, time (3 x 2 x 5) [int32]

```

Exploring

<code>Session.names</code>	Returns the list of names of the objects in the session.
<code>Session.keys(self)</code>	Returns a view on the session's keys.
<code>Session.values(self)</code>	Returns a view on the session's values.
<code>Session.items(self)</code>	Returns a view of the session's items ((key, value) pairs).
<code>Session.summary(self[, template])</code>	Returns a summary of the content of the session.

larray.Session.names

property Session.names

Returns the list of names of the objects in the session. The list is sorted alphabetically and does not follow the internal order.

Returns

list of str

See also:

`Session.keys`

Examples

```

>>> axis1 = Axis("a=a0..a2")
>>> group1 = axis1['a0,a1'] >> 'a01'
>>> arr1, arr2 = ndtest((2, 2)), ndtest(4)
>>> s = Session([('arr2', arr2), ('arr1', arr1), ('group1', group1), ('axis1',
↪axis1)])
>>> # print array's names in the alphabetical order
>>> s.names
['arr1', 'arr2', 'axis1', 'group1']

```

```

>>> # keys() follows the internal order
>>> list(s.keys())
['arr2', 'arr1', 'group1', 'axis1']

```

larray.Session.keys

`Session.keys(self)`

Returns a view on the session's keys.

Returns

View on the session's keys.

See also:

`Session.names`

Examples

```
>>> axis1 = Axis("a=a0..a2")
>>> group1 = axis1['a0,a1'] >> 'a01'
>>> arr1, arr2 = ndtest((2, 2)), ndtest(4)
>>> s = Session([('arr2', arr2), ('arr1', arr1), ('group1', group1), ('axis1',
↪axis1)])
>>> # similar to names by follows the internal order
>>> list(s.keys())
['arr2', 'arr1', 'group1', 'axis1']
```

```
>>> # gives the names of objects in alphabetical order
>>> s.names
['arr1', 'arr2', 'axis1', 'group1']
```

larray.Session.values

`Session.values(self)`

Returns a view on the session's values.

Returns

View on the session's values.

Examples

```
>>> axis1 = Axis("a=a0..a2")
>>> group1 = axis1['a0,a1'] >> 'a01'
>>> arr1, arr2 = ndtest((2, 2)), ndtest(4)
>>> s = Session([('arr2', arr2), ('arr1', arr1), ('group1', group1), ('axis1',
↪axis1)])
>>> # assuming you know the order of objects stored in the session
>>> arr2, arr1, group1, axis1 = s.values()
>>> # otherwise, prefer the following syntax
>>> arr1, arr2, axis1, group1 = s['arr1', 'arr2', 'axis1', 'group1']
>>> arr1
a\b  b0  b1
a0   0   1
a1   2   3
>>> axis1
Axis(['a0', 'a1', 'a2'], 'a')
```

larray.Session.items

`Session.items` (*self*)

Returns a view of the session's items ((key, value) pairs).

Returns

View on the session's items.

Examples

```

>>> axis1 = Axis("a=a0..a2")
>>> group1 = axis1['a0,a1'] >> 'a01'
>>> arr1, arr2 = ndtest((2, 2)), ndtest(4)
>>> # make the test pass on both Windows and Linux
>>> arr1, arr2 = arr1.astype(np.int64), arr2.astype(np.int64)
>>> s = Session([('arr2', arr2), ('arr1', arr1), ('group1', group1), ('axis1',
↪axis1)])
>>> for k, v in s.items():
...     print("{}: {}".format(k, v.info if isinstance(v, Array) else repr(v)))
arr2: 4
  a [4]: 'a0' 'a1' 'a2' 'a3'
dtype: int64
memory used: 32 bytes
arr1: 2 x 2
  a [2]: 'a0' 'a1'
  b [2]: 'b0' 'b1'
dtype: int64
memory used: 32 bytes
group1: a['a0', 'a1'] >> 'a01'
axis1: Axis(['a0', 'a1', 'a2'], 'a')

```

larray.Session.summary

`Session.summary` (*self*, *template=None*)

Returns a summary of the content of the session.

Parameters

template: dict {object type: str} or dict {object type: func} Template describing how items and metadata are summarized. For each object type, it is possible to provide either a string template or a function taking the the key and value of a session item as parameters and returning a string (see examples). A string template contains specific arguments written inside brackets {}. Available arguments are:

- for groups: 'key', 'name', 'axis_name', 'labels' and 'length',
- for axes: 'key', 'name', 'labels' and 'length',
- for arrays: 'key', 'axes_names', 'shape', 'dtype' and 'title',
- for session metadata: 'key', 'value',
- for all other types: 'key', 'value'.

Returns

str Short representation of the content of the session.

Examples

```
>>> axis1 = Axis("a=a0..a2")
>>> group1 = axis1['a0,a1'] >> 'a01'
>>> arr1 = ndtest((2, 2), dtype=np.int64, meta=[('title', 'array 1')])
>>> arr2 = ndtest(4, dtype=np.int64, meta=[('title', 'array 2')])
>>> arr3 = ndtest((3, 2), dtype=np.int64, meta=[('title', 'array 3')])
>>> s = Session([('axis1', axis1), ('group1', group1), ('arr1', arr1), ('arr2',
↳ arr2), ('arr3', arr3)])
>>> s.meta.title = 'my title'
>>> s.meta.author = 'John Smith'
```

Default template

```
>>> print(s.summary())
Metadata:
  title: my title
  author: John Smith
axis1: a ['a0' 'a1' 'a2'] (3)
group1: a['a0', 'a1'] >> a01 (2)
arr1: a, b (2 x 2) [int64]
arr2: a (4) [int64]
arr3: a, b (3 x 2) [int64]
```

Using a specific template

```
>>> def print_array(key, array):
...     axes_names = ', '.join(array.axes.display_names)
...     shape = ' x '.join(str(i) for i in array.shape)
...     return "{} -> {} ({})\n title = {}\n dtype = {}".format(key, axes_names,
↳ shape,
                                                                    array.meta.
↳ title, array.dtype)
>>> template = {Axis: "{key} -> {name} [{labels}] ({length})",
...              Group: "{key} -> {name}: {axis_name}{labels} ({length})",
...              Array: print_array,
...              Metadata: "\t{key} -> {value}"}
>>> print(s.summary(template))
Metadata:
  title -> my title
  author -> John Smith
axis1 -> a ['a0' 'a1' 'a2'] (3)
group1 -> a01: a['a0', 'a1'] (2)
arr1 -> a, b (2 x 2)
  title = array 1
  dtype = int64
arr2 -> a (4)
  title = array 2
  dtype = int64
arr3 -> a, b (3 x 2)
  title = array 3
  dtype = int64
```

Copying

<code>Session.copy(self)</code>	Returns a copy of the session.
---------------------------------	--------------------------------

`larray.Session.copy`

`Session.copy(self)`
Returns a copy of the session.

Testing

<code>Session.element_equals(self, other)</code>	Test if each element (group, axis and array) of the current session equals the corresponding element of another session.
<code>Session.equals(self, other)</code>	Test if all elements (groups, axes and arrays) of the current session are equal to those of another session.

`larray.Session.element_equals`

`Session.element_equals(self, other)`
Test if each element (group, axis and array) of the current session equals the corresponding element of another session.

For arrays, it is equivalent to apply `Array.equals()` with flag `nans_equal=True` to all arrays from two sessions.

Parameters

other [Session] Session to compare with.

Returns

Boolean Array

See also:

`Session.equals`

Notes

Metadata is ignored.

Examples

```
>>> a = Axis('a=a0..a2')
>>> a01 = a['a0,a1'] >> 'a01'
>>> s1 = Session([('a', a), ('a01', a01), ('arr1', ndtest(2)), ('arr2', ndtest((2,
↪ 2))))])
>>> s2 = Session([('a', a), ('a01', a01), ('arr1', ndtest(2)), ('arr2', ndtest((2,
↪ 2))))])
```

Identical sessions

```
>>> s1.element_equals(s2)
name      a    a01   arr1   arr2
      True  True   True   True
```

Different value(s) between two arrays

```
>>> s2.arr1['a1'] = 0
>>> s1.element_equals(s2)
name      a    a01   arr1   arr2
      True  True  False   True
```

Different label(s)

```
>>> s2.arr2 = ndtest("b=b0,b1; a=a0,a1")
>>> s2.a = Axis('a=a0,a1')
>>> s1.element_equals(s2)
name      a    a01   arr1   arr2
      False True  False  False
```

Extra/missing objects

```
>>> s2.arr3 = ndtest((3, 3))
>>> del s2.a
>>> s1.element_equals(s2)
name      a    a01   arr1   arr2   arr3
      False True  False  False  False
```

larray.Session.equals

`Session.equals(self, other)`

Test if all elements (groups, axes and arrays) of the current session are equal to those of another session.

Parameters

other [Session] Session to compare with.

Returns

True if elements of both sessions are all equal, False otherwise.

See also:

[`Session.element_equals`](#)

Notes

Metadata is ignored.

Examples

```
>>> a = Axis('a=a0..a2')
>>> a01 = a['a0,a1'] >> 'a01'
>>> s1 = Session([('a', a), ('a01', a01), ('arr1', ndtest(2)), ('arr2', ndtest((2,
↪ 2))))])
>>> s2 = Session([('a', a), ('a01', a01), ('arr1', ndtest(2)), ('arr2', ndtest((2,
↪ 2))))])
```

Identical sessions

```
>>> s1.equals(s2)
True
```

Different value(s) between two arrays

```
>>> s2.arr1['a1'] = 0
>>> s1.equals(s2)
False
```

Different label(s)

```
>>> s2.arr2 = ndtest("b=b0,b1; a=a0,a1")
>>> s2.a = Axis('a=a0,a1')
>>> s1.equals(s2)
False
```

Extra/missing axis(es), group(s), array(s)

```
>>> s2.arr3 = ndtest((3, 3))
>>> del s2.a
>>> s1.equals(s2)
False
```

Selecting

`Session.get(self, key[, default])`

Returns the object corresponding to the key.

`larray.Session.get`

`Session.get` (*self*, *key*, *default=None*)

Returns the object corresponding to the key. If the key doesn't correspond to any object, a default one can be returned.

Parameters

key [str] Name of the object.

default [object, optional] Returned object if the key doesn't correspond to any object of the

current session.

Returns

object Object corresponding to the given key or a default one if not found.

Examples

```
>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
>>> arr = s.get('arr1')
>>> arr
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
a2    6   7   8
>>> arr = s.get('arr4', zeros('a=a0,a1;b=b0,b1', dtype=int))
>>> arr
a\b  b0  b1
a0    0   0
a1    0   0
```

Modifying

<code>Session.add(self, *args, **kwargs)</code>	Adds objects to the current session.
<code>Session.update(self, other)</code>	Update the session with the key/value pairs from other or passed keyword arguments, overwriting existing keys.
<code>Session.get(self, key[, default])</code>	Returns the object corresponding to the key.
<code>Session.apply(self, func, *args, **kwargs)</code>	Apply function <i>func</i> on elements of the session and return a new session.
<code>Session.transpose(self, *args)</code>	Reorder axes of arrays in session, ignoring missing axes for each array.

larray.Session.add

`Session.add(self, *args, **kwargs)`
Adds objects to the current session.

Parameters

***args** [list of object] Objects to add. Objects must have an attribute ‘name’.

****kwargs** [dict of {str: object}] Objects to add written as name=array, ...

Examples

```
>>> s = Session()
>>> axis1, axis2 = Axis('x=x0..x2'), Axis('y=y0..y2')
>>> arr1, arr2, arr3 = ndtest((2, 2)), ndtest(4), ndtest((3, 2))
>>> s.add(axis1, axis2, arr1=arr1, arr2=arr2, arr3=arr3)
>>> # print item's names in sorted order
>>> s.names
['arr1', 'arr2', 'arr3', 'x', 'y']
```

larray.Session.update

`Session.update` (*self*, *other=None*, ***kwargs*)

Update the session with the key/value pairs from *other* or passed keyword arguments, overwriting existing keys. Note that the session is updated inplace and no new Session object is returned.

Parameters

other: Session or dict-like object or iterable with key/value pairs Object containing key/value pairs to add or modify.

****kwargs:** If keyword arguments are specified, the session is then updated with those key/value pairs (e.g.: `ses.update(pop=pop, births=births, deaths=deaths)`).

Examples

```
>>> x, y = Axis('x=x0..x2'), Axis('y=y0..y3')
>>> arr1 = ndtest((x, y))
>>> arr2 = ndtest(x)
>>> s = Session(x=x, y=y, arr1=arr1, arr2=arr2)
>>> # print item's names in sorted order
>>> s.names
['arr1', 'arr2', 'x', 'y']
>>> s.arr2
x  x0  x1  x2
   0   1   2
```

```
>>> # new axis and array
>>> z = Axis('z=z0..z2')
>>> arr3 = ndtest((x, z))
>>> # arr2 is modified
>>> arr2_modified = arr2.set_axes('x', z)
```

Passing another session

```
>>> s2 = Session(z=z, arr2=arr2_modified, arr3=arr3)
>>> s.names
['arr1', 'arr2', 'x', 'y']
>>> s.arr2
x  x0  x1  x2
   0   1   2
>>> s.update(s2)
>>> # new items have been added to the session 's'
>>> s.names
```

(continues on next page)

(continued from previous page)

```
['arr1', 'arr2', 'arr3', 'x', 'y', 'z']
>>> # and array 'arr2' has been updated
>>> s.arr2
z  z0  z1  z2
   0   1   2
```

Passing a dictionary

```
>>> s = Session(x=x, y=y, arr1=arr1, arr2=arr2)
>>> s.names
['arr1', 'arr2', 'x', 'y']
>>> s.arr2
x  x0  x1  x2
   0   1   2
>>> d = {'z': z, 'arr2': arr2_modified, 'arr3': arr3}
>>> s.update(d)
>>> s.names
['arr1', 'arr2', 'arr3', 'x', 'y', 'z']
>>> s.arr2
z  z0  z1  z2
   0   1   2
```

Passing an iterable with key/value pairs

```
>>> s = Session(x=x, y=y, arr1=arr1, arr2=arr2)
>>> s.names
['arr1', 'arr2', 'x', 'y']
>>> s.arr2
x  x0  x1  x2
   0   1   2
>>> i = [('z', z), ('arr2', arr2_modified), ('arr3', arr3)]
>>> s.update(i)
>>> s.names
['arr1', 'arr2', 'arr3', 'x', 'y', 'z']
>>> s.arr2
z  z0  z1  z2
   0   1   2
```

Passing keyword arguments

```
>>> s = Session(x=x, y=y, arr1=arr1, arr2=arr2)
>>> s.names
['arr1', 'arr2', 'x', 'y']
>>> s.arr2
x  x0  x1  x2
   0   1   2
>>> s.update(z=z, arr2=arr2_modified, arr3=arr3)
>>> s.names
['arr1', 'arr2', 'arr3', 'x', 'y', 'z']
>>> s.arr2
z  z0  z1  z2
   0   1   2
```

larray.Session.apply

`Session.apply(self, func, *args, **kwargs)`

Apply function *func* on elements of the session and return a new session.

Parameters

func [function] Function to apply to each element of the session. It should take a single *element* argument and return a single value.

***args** [any] Any extra arguments are passed to the function

kind [type or tuple of types, optional] Type(s) of elements *func* will be applied to. Other elements will be left intact. Use 'kind=object' to apply to all kinds of objects. Defaults to Array.

****kwargs** [any] Any extra keyword arguments are passed to the function

Returns

Session A new session containing all processed elements

Examples

```
>>> arr1 = ndtest(2)
>>> arr1
a  a0  a1
   0   1
>>> arr2 = ndtest(3)
>>> arr2
a  a0  a1  a2
   0   1   2
>>> sess1 = Session([('arr1', arr1), ('arr2', arr2)])
>>> sess1
Session(arr1, arr2)
>>> def increment(array):
...     return array + 1
>>> sess2 = sess1.apply(increment)
>>> sess2.arr1
a  a0  a1
   1   2
>>> sess2.arr2
a  a0  a1  a2
   1   2   3
```

You may also pass extra arguments or keyword arguments to the function

```
>>> def change(array, increment=1, multiplier=1):
...     return (array + increment) * multiplier
>>> sess2 = sess1.apply(change, 2, 2)
>>> sess2 = sess1.apply(change, 2, multiplier=2)
>>> sess2.arr1
a  a0  a1
   4   6
>>> sess2.arr2
a  a0  a1  a2
   4   6   8
```

`larray.Session.transpose`

`Session.transpose(self, *args)`

Reorder axes of arrays in session, ignoring missing axes for each array.

Parameters

***args** Accepts either a tuple of axes specs or axes specs as **args*. Omitted axes keep their order. Use ... to avoid specifying intermediate axes. Axes missing in an array are ignored.

Returns

Session Session with each array with reordered axes where appropriate.

See also:

[`Array.transpose`](#)

Examples

Let us create a test session and a small helper function to display sessions as a short summary.

```
>>> arr1 = ndtest((2, 2, 2))
>>> arr2 = ndtest((2, 2))
>>> sess = Session([('arr1', arr1), ('arr2', arr2)])
>>> def print_summary(s):
...     print(s.summary({Array: "{key} -> {axes_names}"}))
>>> print_summary(sess)
arr1 -> a, b, c
arr2 -> a, b
```

Put 'b' axis in front of all arrays

```
>>> print_summary(sess.transpose('b'))
arr1 -> b, a, c
arr2 -> b, a
```

Axes missing on an array are ignored ('c' for arr2 in this case)

```
>>> print_summary(sess.transpose('c', 'b'))
arr1 -> c, b, a
arr2 -> b, a
```

Use ... to move axes to the end

```
>>> print_summary(sess.transpose(..., 'a'))
arr1 -> b, c, a
arr2 -> b, a
```

Filtering/Cleaning

<code>Session.filter(self[, pattern, kind])</code>	Returns a new session with objects which match some criteria.
<code>Session.compact(self[, display])</code>	Detects and removes “useless” axes (ie axes for which values are constant over the whole axis) for all array objects in session

larray.Session.filter

`Session.filter(self, pattern=None, kind=None)`

Returns a new session with objects which match some criteria.

Parameters

pattern [str, optional] Only keep arrays whose key match *pattern*.

- ? matches any single character
- * matches any number of characters
- [seq] matches any character in seq
- [!seq] matches any character not in seq

kind [(tuple of) type, optional] Only keep objects which are instances of type(s) *kind*.

Returns

Session The filtered session.

Examples

```
>>> axis = Axis('a=a0..a2')
>>> group = axis['a0,a1'] >> 'a01'
>>> test1, zero1 = ndtest((2, 2)), zeros((3, 2))
>>> s = Session([('test1', test1), ('zero1', zero1), ('axis', axis), ('group',
↪group)])
```

Filter using a pattern argument

```
>>> # get all items with names ending with '1'
>>> s.filter(pattern='*1').names
['test1', 'zero1']
```

```
>>> # get all items with names starting with letter in range a-k
>>> s.filter(pattern='[a-k]*').names
['axis', 'group']
```

Filter using kind argument

```
>>> s.filter(kind=Axis).names
['axis']
>>> s.filter(kind=(Axis, Group)).names
['axis', 'group']
```

larray.Session.compact

`Session.compact` (*self*, *display=False*)

Detects and removes “useless” axes (ie axes for which values are constant over the whole axis) for all array objects in session

Parameters

display [bool, optional] Whether or not to display a message for each array that is compacted

Returns

Session A new session containing all compacted arrays

Examples

```
>>> arr1 = sequence('b=b0..b2', ndtest(3), zeros_like(ndtest(3)))
>>> arr1
a\b  b0  b1  b2
a0    0   0   0
a1    1   1   1
a2    2   2   2
>>> compact_ses = Session(arr1=arr1).compact(display=True)
arr1 was constant over {b}
>>> compact_ses.arr1
a  a0  a1  a2
   0   1   2
```

Load/Save

<code>Session.load</code> (<i>self</i> , <i>fname</i> [, <i>names</i> , <i>engine</i> , ...])	Load Array objects from a file, or several .csv files (all formats).
<code>Session.save</code> (<i>self</i> , <i>fname</i> [, <i>names</i> , <i>engine</i> , ...])	Dumps Array objects from the current session to a file (all formats).
<code>Session.to_csv</code> (<i>self</i> , <i>fname</i> [, <i>names</i> , <i>display</i>])	Dumps Array objects from the current session to CSV files.
<code>Session.to_excel</code> (<i>self</i> , <i>fname</i> [, <i>names</i> , ...])	Dumps Array objects from the current session to an Excel file.
<code>Session.to_hdf</code> (<i>self</i> , <i>fname</i> [, <i>names</i> , ...])	Dumps Array, Axis and Group objects from the current session to an HDF file.
<code>Session.to_pickle</code> (<i>self</i> , <i>fname</i> [, <i>names</i> , ...])	Dumps Array, Axis and Group objects from the current session to a file using pickle.

larray.Session.load

`Session.load(self, fname, names=None, engine='auto', display=False, **kwargs)`

Load Array objects from a file, or several .csv files (all formats). Load also Axis and Group objects from a file (HDF and pickle formats).

WARNING: never load a file using the pickle engine (.pkl or .pickle) from an untrusted source, as it can lead to arbitrary code execution.

Parameters

fname [str] This can be either the path to a single file, a path to a directory containing .csv files or a pattern representing several .csv files.

names [list of str, optional] List of objects to load. If *fname* is None, list of paths to CSV files. Defaults to all valid objects present in the file/directory.

engine [{ 'auto', 'pandas_csv', 'pandas_hdf', 'pandas_excel', 'xlwings_excel', 'pickle' }, optional] Load using *engine*. Defaults to 'auto' (use default engine for the format guessed from the file extension).

display [bool, optional] Whether or not to display which file is being worked on. Defaults to False.

Examples

In one module:

```

>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
>>> # metadata
>>> s.meta.title = 'my title'
>>> s.meta.author = 'John Smith'
>>> # save the session in an HDF5 file
>>> s.save('input.h5')

```

In another module: load the whole session

```

>>> # the load method is automatically called when passing
>>> # the path of file to the Session constructor
>>> s = Session('input.h5')
>>> s
Session(a, b, a01, arr1, arr2)
>>> s.meta
title: my title
author: John Smith

```

Load only some objects

```

>>> s = Session()
>>> s.load('input.h5', ['a', 'b', 'arr1', 'arr2'])
>>> a, b, arr1, arr2 = s['a', 'b', 'arr1', 'arr2']

```

(continues on next page)

(continued from previous page)

```
>>> # only if you know the order of arrays stored in session
>>> a, b, a01, arr1, arr2 = s.values()
```

Using .csv files (assuming the same session as above)

```
>>> s.save('data')
>>> s = Session()
>>> # load all .csv files starting with "output" in the data directory
>>> s.load('data')
>>> # or only arrays (i.e. all CSV files starting with 'arr')
>>> s.load('data/arr*.csv')
```

larray.Session.save

`Session.save(self, fname, names=None, engine='auto', overwrite=True, display=False, **kwargs)`

Dumps Array objects from the current session to a file (all formats). Dumps also Axis and Group objects from the current session to a file (HDF and pickle format).

Parameters

fname [str] Path of the file for the dump. If objects are saved in CSV files, the path corresponds to a directory.

names [list of str or None, optional] List of names of Array/Axis/Group objects to dump. If *fname* is None, list of paths to CSV files. Defaults to all objects present in the Session.

engine [{ 'auto', 'pandas_csv', 'pandas_hdf', 'pandas_excel', 'xlwings_excel', 'pickle' }, optional] Dump using *engine*. Defaults to 'auto' (use default engine for the format guessed from the file extension).

overwrite: bool, optional Whether or not to overwrite an existing file, if any. Ignored for CSV files and 'pandas_excel' engine. If False, file is updated. Defaults to True.

display [bool, optional] Whether or not to display which file is being worked on. Defaults to False.

Examples

```
>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
>>> # metadata
>>> s.meta.title = 'my title'
>>> s.meta.author = 'John Smith'
```

Save all objects

```
>>> s.save('output.h5')
```

Save only some objects

```
>>> s.save('output.h5', ['a', 'b', 'arr1'])
```

Update file

```
>>> arr1, arr4 = ndtest((3, 3)), ndtest((2, 3))
>>> s2 = Session([('arr1', arr1), ('arr4', arr4)])
>>> # replace arr1 and add arr4 in file output.h5
>>> s2.save('output.h5', overwrite=False)
```

larray.Session.to_csv

`Session.to_csv(self, fname, names=None, display=False, **kwargs)`

Dumps Array objects from the current session to CSV files.

Parameters

fname [str] Path for the directory that will contain CSV files.

names [list of str or None, optional] Names of Array objects to dump. Defaults to all Array objects present in the Session.

display [bool, optional] Whether or not to display which file is being worked on. Defaults to False.

Notes

- each array is saved in a separate file
- all session metadata is saved in the same CSV file named `__metadata__.csv`

Examples

```
>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
>>> # metadata
>>> s.meta.title = 'my title'
>>> s.meta.author = 'John Smith'
```

Save all arrays

```
>>> s.to_csv('./Output')
```

Save only some arrays

```
>>> s.to_csv('./Output', ['a', 'b', 'arr1'])
```

`larray.Session.to_excel`

`Session.to_excel` (*self*, *fname*, *names=None*, *overwrite=True*, *display=False*, ***kwargs*)

Dumps Array objects from the current session to an Excel file.

Parameters

fname [str] Path of the file for the dump.

names [list of str or None, optional] Names of Array objects to dump. Defaults to all Array objects present in the Session.

overwrite: bool, optional Whether or not to overwrite an existing file, if any. If False, file is updated. Defaults to True.

display [bool, optional] Whether or not to display which file is being worked on. Defaults to False.

Notes

- each array is saved in a separate sheet
- all session metadata is saved in the same sheet named `__metadata__`

Examples

```
>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
>>> # metadata
>>> s.meta.title = 'my title'
>>> s.meta.author = 'John Smith'
```

Save all arrays

```
>>> s.to_excel('output.xlsx')
```

Save only some objects

```
>>> s.to_excel('output.xlsx', ['a', 'b', 'arr1'])
```

larray.Session.to_hdf

`Session.to_hdf(self, fname, names=None, overwrite=True, display=False, **kwargs)`

Dumps Array, Axis and Group objects from the current session to an HDF file.

Parameters

fname [str] Path of the file for the dump.

names [list of str or None, optional] Names of Array/Axis/Group objects to dump. Defaults to all objects present in the Session.

overwrite: bool, optional Whether or not to overwrite an existing file, if any. If False, file is updated. Defaults to True.

display [bool, optional] Whether or not to display which file is being worked on. Defaults to False.

Examples

```

>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
>>> # metadata
>>> s.meta.title = 'my title'
>>> s.meta.author = 'John Smith'

```

Save all arrays

```

>>> s.to_hdf('output.h5')

```

Save only some objects

```

>>> s.to_hdf('output.h5', ['a', 'b', 'arr1'])

```

larray.Session.to_pickle

`Session.to_pickle(self, fname, names=None, overwrite=True, display=False, **kwargs)`

Dumps Array, Axis and Group objects from the current session to a file using pickle.

WARNING: never load a pickle file (.pkl or .pickle) from an untrusted source, as it can lead to arbitrary code execution.

Parameters

fname [str] Path for the dump.

names [list of str or None, optional] Names of Array/Axis/Group objects to dump. Defaults to all objects present in the Session.

overwrite: bool, optional Whether or not to overwrite an existing file, if any. If False, file is updated. Defaults to True.

display [bool, optional] Whether or not to display which file is being worked on. Defaults to False.

Examples

```
>>> # axes
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> # groups
>>> a01 = a['a0,a1'] >> 'a01'
>>> # arrays
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
>>> # metadata
>>> s.meta.title = 'my title'
>>> s.meta.author = 'John Smith'
```

Save all arrays

```
>>> s.to_pickle('output.pkl')
```

Save only some objects

```
>>> s.to_pickle('output.pkl', ['a', 'b', 'arr1'])
```

4.3.14 Editor

<code>view([obj, title, depth])</code>	Opens a new viewer window.
<code>edit([obj, title, minvalue, maxvalue, ...])</code>	Opens a new editor window.
<code>debug([depth])</code>	Opens a new debug window.
<code>compare(*args, **kwargs)</code>	Opens a new comparator window, comparing arrays or sessions.
<code>run_editor_on_exception([root_path, ...])</code>	Runs the editor when an unhandled exception (a fatal error) happens.

larray.view

`larray.view(obj=None, title="", depth=0)`

Opens a new viewer window. Arrays are loaded in readonly mode and their content cannot be modified.

Parameters

obj [np.ndarray, Array, Session, dict or str, optional] Object to visualize. If string, array(s) will be loaded from the file given as argument. Defaults to the collection of all local variables where the function was called.

title [str, optional] Title for the current object. Defaults to the name of the first object found in the caller namespace which corresponds to *obj* (it will use a combination of the 3 first names if several names correspond to the same object).

depth [int, optional] Stack depth where to look for variables. Defaults to 0 (where this function was called).

Examples

```
>>> a1 = ndtest(3)
>>> a2 = ndtest(3) + 1
>>> # will open a viewer showing all the arrays available at this point
>>> # (a1 and a2 in this case)
>>> view()
>>> # will open a viewer showing only a1
>>> view(a1)
```

larray.edit

`larray.edit(obj=None, title="", minvalue=None, maxvalue=None, readonly=False, depth=0)`

Opens a new editor window.

Parameters

obj [np.ndarray, Array, Session, dict, str or REOPEN_LAST_FILE, optional] Object to visualize. If string, array(s) will be loaded from the file given as argument. Passing the constant REOPEN_LAST_FILE loads the last opened file. Defaults to the collection of all local variables where the function was called.

title [str, optional] Title for the current object. Defaults to the name of the first object found in the caller namespace which corresponds to *obj* (it will use a combination of the 3 first names if several names correspond to the same object).

minvalue [scalar, optional] Minimum value allowed.

maxvalue [scalar, optional] Maximum value allowed.

readonly [bool, optional] Whether or not editing array values is forbidden. Defaults to False.

depth [int, optional] Stack depth where to look for variables. Defaults to 0 (where this function was called).

Examples

```
>>> a1 = ndtest(3)
>>> a2 = ndtest(3) + 1
>>> # will open an editor with all the arrays available at this point
>>> # (a1 and a2 in this case)
>>> edit()
>>> # will open an editor for a1 only
>>> edit(a1)
```

larray.debug

`larray.debug(depth=0)`

Opens a new debug window.

Parameters

depth [int, optional] Stack depth where to look for variables. Defaults to 0 (where this function was called).

larray.compare

`larray.compare(*args, **kwargs)`

Opens a new comparator window, comparing arrays or sessions.

Parameters

***args** [Arrays or Sessions] Arrays or sessions to compare.

title [str, optional] Title for the window. Defaults to ‘’.

names [list of str, optional] Names for arrays or sessions being compared. Defaults to the name of the first objects found in the caller namespace which correspond to the passed objects.

depth [int, optional] Stack depth where to look for variables. Defaults to 0 (where this function was called).

display_caller_info: bool, optional Whether or not to display the filename and line number where the Editor has been called. Defaults to True.

rtol [float or int, optional] The relative tolerance parameter (see Notes). Defaults to 0.

atol [float or int, optional] The absolute tolerance parameter (see Notes). Defaults to 0.

nans_equal [boolean, optional] Whether or not to consider NaN values at the same positions in the two arrays as equal. By default, an array containing NaN values is never equal to another array, even if that other array also contains NaN values at the same positions. The reason is that a NaN value is different from *anything*, including itself. Defaults to True.

Notes

For finite values, the following equation is used to test whether two values are equal:

$$\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$$

Examples

```
>>> a1 = ndtest(3)
>>> a2 = ndtest(3) + 1
>>> compare(a1, a2, title='first comparison')
>>> compare(a1 + 1, a2, title='second comparison', names=['a1+1', 'a2'])
```

larray.run_editor_on_exception

`larray.run_editor_on_exception(root_path=None, usercode_traceback=True, usercode_frame=True)`

Runs the editor when an unhandled exception (a fatal error) happens.

Parameters

root_path [str, optional] Defaults to None (the directory of the main script).

usercode_traceback [bool, optional] Whether or not to show only the part of the traceback (error log) which corresponds to the user code. Otherwise, it will show the complete traceback, including code inside libraries. Defaults to True.

usercode_frame [bool, optional] Whether or not to start the debug window in the frame corresponding to the user code. This argument is ignored (it is always True) if `usercode_traceback` is True. Defaults to True.

Notes

sets `sys.excepthook`

4.3.15 Random

<code>random.randint(low[, high, axes, dtype, meta])</code>	Return random integers from <i>low</i> (inclusive) to <i>high</i> (exclusive).
<code>random.normal([loc, scale, axes, meta])</code>	Draw random samples from a normal (Gaussian) distribution.
<code>random.uniform([low, high, axes, meta])</code>	Draw samples from a uniform distribution.
<code>random.permutation(x[, axis])</code>	Randomly permute a sequence along an axis, or return a permuted range.
<code>random.choice([choices, axes, replace, p, meta])</code>	Generates a random sample from given choices

larray.random.randint

`larray.random.randint` (*low*, *high=None*, *axes=None*, *dtype='l'*, *meta=None*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the “discrete uniform” distribution of the specified dtype in the “half-open” interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Parameters

low [int] Lowest (signed) integer to be drawn from the distribution (unless *high=None*, in which case this parameter is one above the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if *high=None*).

axes [int, tuple of int, str, Axis or tuple/list/AxisCollection of Axis, optional] Axes (or shape) of the resulting array. If *axes* is None (the default), a single value is returned. Otherwise, if the resulting axes have a shape of, e.g., (*m*, *n*, *k*), then *m* * *n* * *k* samples are drawn.

dtype [data-type, optional] Desired dtype of the result. All dtypes are determined by their name, i.e., ‘int64’, ‘int’, etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is ‘np.int’.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array

Examples

Generate a single int between 0 and 9, inclusive:

```
>>> la.random.randint(10)
6
```

Generate an array of 10 ints between 1 and 5, inclusive:

```
>>> la.random.randint(1, 6, 10)
{0}*  0  1  2  3  4  5  6  7  8  9
      1  1  5  1  5  4  3  4  2  1
```

Generate a 2 x 3 array of ints between 0 and 4, inclusive:

```
>>> la.random.randint(5, axes=(2, 3))
{0}*\\{1}*  0  1  2
            0  4  4  1
            1  1  2  2
>>> la.random.randint(5, axes='a=a0,a1;b=b0..b2')
a\\b  b0  b1  b2
a0    0   3   1
a1    4   0   1
```

larray.random.normal

`larray.random.normal` (*loc=0.0, scale=1.0, axes=None, meta=None*)

Draw random samples from a normal (Gaussian) distribution.

Its probability density function is often called the bell curve because of its characteristic shape (see the example below)

Parameters

loc [float or array_like of floats] Mean (“centre”) of the distribution.

scale [float or array_like of floats] Standard deviation (spread or “width”) of the distribution.

axes [int, tuple of int, str, Axis or tuple/list/AxisCollection of Axis, optional] Minimum axes the resulting array must have. Defaults to None. The resulting array axes will be the union of those mentioned in *axes* and those of *loc* and *scale*. If *loc* and *scale* are scalars and *axes* is None, a single value is returned. Otherwise, if the resulting axes have a shape of, e.g., (*m*, *n*, *k*), then *m* * *n* * *k* samples are drawn.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array or scalar Drawn samples from the parameterized normal distribution.

Notes

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

The probability density function for the Gaussian distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2], is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2 , is called the variance.

The function has its peak at the mean, and its “spread” increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]). This implies that *la.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

References

[1], [2]

Examples

Generate a 2 x 3 array with numbers drawn from the distribution:

```
>>> la.random.normal(0, 1, axes=(2, 3))
{0}*{1}*
      0      1      2
0  0.3564325741877542  0.8944149721039006  1.7206904920773107
1  0.6904447654719367 -0.09395966570976753  0.185136309092257
```

With named and labelled axes

```
>>> la.random.normal(0, 1, axes='a=a0,a1;b=b0..b2')
a\b      b0      b1      b2
a0  2.3096106652701827 -0.4269082412118316 -1.0862791566867225
a1  0.8598817639620348 -2.386411240813283  0.10116503197279443
```

With varying loc and scale (each depending on a different axis)

```
>>> a = la.Axis('a=a0,a1')
>>> b = la.Axis('b=b0..b2')
>>> mu = la.sequence(a, initial=5, inc=5)
>>> mu
a  a0  a1
   5  10
>>> sigma = la.sequence(b, initial=1)
>>> sigma
b  b0  b1  b2
   1  2  3
>>> la.random.normal(mu, sigma)
a\b      b0      b1      b2
a0  5.939369790854615  2.5043856460438403  8.33560126941519
a1  10.759526714752091  10.093213549397403  11.705881778249683
```

Draw 1000 samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> sample = la.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - la.mean(sample)) < 0.01
True
>>> abs(sigma - la.std(sample, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(sample, 30, normed=True)
>>> pdf = 1 / (sigma * la.sqrt(2 * la.pi)) \
...         * la.exp(-(bins - mu) ** 2 / (2 * sigma ** 2))
>>> _ = plt.plot(bins, pdf, linewidth=2, color='r')
>>> plt.show()
```

larray.random.uniform

`larray.random.uniform` (*low*=0.0, *high*=1.0, *axes*=None, *meta*=None)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [*low*, *high*) (includes *low*, but excludes *high*). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

low [float or array_like of floats, optional] Lower boundary of the output interval. All values generated will be greater than or equal to *low*. Defaults to 0.0.

high [float or array_like of floats, optional] Upper boundary of the output interval. All values generated will be less than *high*. Defaults to 1.0.

axes [int, tuple of int, str, Axis or tuple/list/AxisCollection of Axis, optional] Minimum axes the resulting array must have. Defaults to None. The resulting array axes will be the union of those mentioned in *axes* and those of *low* and *high*. If *low* and *high* are scalars and *axes* is None, a single value is returned. Otherwise, if the resulting axes have a shape of, e.g., (*m*, *n*, *k*), then *m* * *n* * *k* samples are drawn.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array or scalar Drawn samples from the parameterized uniform distribution.

See also:

[`randint`](#) Discrete uniform distribution, yielding integers.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b-a}$$

anywhere within the interval $[a, b)$, and zero elsewhere.

When `high == low`, values of `low` will be returned. If `high < low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

Examples

Generate a single sample from the distribution:

```
>>> la.random.uniform()
0.4616049008844396
```

Generate a 2 x 3 array with numbers drawn from the distribution:

```
>>> la.random.uniform(0, 5, axes=(2, 3))
{0}*{1}*          0          1          2
0  3.4951791043804192  3.888533056628081  4.347461073315136
1  2.146211610940853  0.509146487437932  2.790852715735223
```

With named and labelled axes

```
>>> la.random.uniform(1, 2, axes='a=a0,a1;b=b0..b2')
a\b          b0          b1          b2
a0  1.4167729850467825  1.6953091052066793  1.2321770607672526
a1  1.4386221912579358  1.8480607144284926  1.1726213637670433
```

With varying low and high (each depending on a different axis)

```
>>> a = la.Axis('a=a0,a1')
>>> b = la.Axis('b=b0..b2')
>>> low = la.sequence(a)
>>> low
a  a0  a1
   0   1
>>> high = la.sequence(b, initial=1, inc=0.5)
>>> high
b  b0  b1  b2
   1.0 1.5 2.0
>>> la.random.uniform(low, high)
a\b          b0          b1          b2
a0  0.44608671494167573  0.948315996350121  1.74189664009661
a1          1.0  1.1099944474264194  1.1362792569316835
```

Draw 1000 samples from the distribution:

```
>>> s = la.random.uniform(-1, 0, 1000)
```

All values are within the given interval:

```
>>> la.all(s >= -1)
True
>>> la.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> _ = plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

larray.random.permutation

`larray.random.permutation(x, axis=0)`

Randomly permute a sequence along an axis, or return a permuted range.

Parameters

x [int or array_like] If *x* is an integer, randomly permute `sequence(x)`. If *x* is an array, returns a randomly shuffled copy.

axis [int, str or Axis, optional] Axis along which to permute. Defaults to the first axis.

Returns

Array Permuted sequence or array range.

Examples

```
>>> la.random.permutation(10)
{0}* 0 1 2 3 4 5 6 7 8 9
      6 8 0 9 4 7 1 5 3 2
>>> la.random.permutation([1, 4, 9, 12, 15])
{0}* 0 1 2 3 4
      1 15 12 9 4
>>> la.random.permutation(la.ndtest(5))
a  a3 a1 a2 a4 a0
   3 1 2 4 0
>>> arr = la.ndtest((3, 3))
>>> la.random.permutation(arr)
a\b  b0 b1 b2
a1   3 4 5
a2   6 7 8
a0   0 1 2
>>> la.random.permutation(arr, axis='b')
a\b  b1 b2 b0
a0   1 2 0
a1   4 5 3
a2   7 8 6
```

larray.random.choice

`larray.random.choice` (*choices=None, axes=None, replace=True, p=None, meta=None*)

Generates a random sample from given choices

Parameters

choices [1-D array-like or int, optional] Values to choose from. If an array, a random sample is generated from its elements. If an int *n*, the random sample is generated as if choices was `la.sequence(n)` If *p* is a 1-D Array, choices are taken from its axis.

axes [int, tuple of int, str, Axis or tuple/list/AxisCollection of Axis, optional] Axes (or shape) of the resulting array. If *axes* is `None` (the default), a single value is returned. Otherwise, if the resulting axes have a shape of, e.g., (m, n, k) , then $m * n * k$ samples are drawn.

replace [boolean, optional] Whether the sample is with or without replacement.

p [array-like, optional] The probabilities associated with each entry in choices. If *p* is a 1-D Array, choices are taken from its axis labels. If *p* is an N-D Array, each cell represents the probability that the combination of labels will occur. If not given the sample assumes a uniform distribution over all entries in choices.

meta [list of pairs or dict or OrderedDict or Metadata, optional] Metadata (title, description, author, creation_date, ...) associated with the array. Keys must be strings. Values must be of type string, int, float, date, time or datetime.

Returns

Array or scalar The generated random samples with given *axes* (or shape).

Raises

ValueError If *choices* is an int and less than zero, if *choices* or *p* are not 1-dimensional, if *choices* is an array-like of size 0, if *p* is not a vector of probabilities, if *choices* and *p* have different lengths, or if *replace=False* and the sample size is greater than the population size.

See also:

[*randint, permutation*](#)

Examples

Generate one random value out of given choices (each choice has the same probability of occurring):

```
>>> la.random.choice(['hello', 'world', '!'])
hello
```

With given probabilities:

```
>>> la.random.choice(['hello', 'world', '!'], p=[0.1, 0.8, 0.1])
world
```

Generate a 2 x 3 array with given axes and values drawn from the given choices using given probabilities:

```
>>> la.random.choice([5, 10, 15], p=[0.3, 0.5, 0.2], axes='a=a0,a1;b=b0..b2')
a\b  b0  b1  b2
a0   15  10  10
a1   10   5  10
```

Same as above with labels and probabilities given as a one dimensional Array

```
>>> proba = Array([0.3, 0.5, 0.2], Axis([5, 10, 15], 'outcome'))
>>> proba
outcome    5    10    15
          0.3  0.5  0.2
>>> choice(p=proba, axes='a=a0,a1;b=b0..b2')
a\b  b0  b1  b2
a0   10  15   5
a1   10   5  10
```

Generate a uniform random sample of size 3 from la.sequence(5):

```
>>> la.random.choice(5, 3)
{0}*  0  1  2
      3  2  0
>>> # This is equivalent to la.random.randint(0, 5, 3)
```

Generate a non-uniform random sample of size 3 from the given choices without replacement:

```
>>> la.random.choice(['hello', 'world', '!'], 3, replace=False, p=[0.1, 0.6, 0.3])
{0}*    0  1    2
      world !  hello
```

Using an N-dimensional array as probabilities:

```
>>> proba = Array([[0.15, 0.25, 0.10],
...                [0.20, 0.10, 0.20]], 'a=a0,a1;b=b0..b2')
>>> proba
a\b    b0    b1    b2
a0   0.15  0.25  0.1
a1   0.2   0.1   0.2
>>> choice(p=proba, axes='draw=d0..d5')
draw\axis  a    b
          d0  a1  b2
          d1  a1  b1
          d2  a0  b1
          d3  a0  b0
          d4  a1  b2
          d5  a0  b1
```

4.3.16 Constants

<i>nan</i>	NaN (Not a Number)
<i>inf</i>	∞ (infinite)
<i>pi</i>	π
<i>e</i>	e
<i>euler_gamma</i>	Euler's γ

larray.core.constants.nan

```
larray.core.constants.nan = nan
```

NaN (Not a Number)

larray.core.constants.inf

```
larray.core.constants.inf = inf
```

∞ (infinite)

larray.core.constants.pi

```
larray.core.constants.pi = 3.141592653589793
```

π

larray.core.constants.e

```
larray.core.constants.e = 2.718281828459045
```

e

larray.core.constants.euler_gamma

```
larray.core.constants.euler_gamma = 0.5772156649015329
```

Euler's γ

INDICES AND TABLES

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

6.1 Change log

6.1.1 Version 0.32

Released on 2019-11-17.

CORE

Syntax changes

- renamed the `LArray` class to `Array` (closes [issue 611](#)).

Backward incompatible changes

- Because it was broken, the possibility to dump and load `Axis` and `Group` objects contained in a session has been removed for the CSV and Excel formats. Fixing it would have taken too much time considering it is very rarely used (no one complains it was broken) so the decision to remove it was taken. However, this is still possible using the HDF format. Closes [issue 815](#).

Miscellaneous improvements

- conda channel to install or update the `larray`, `larray-editor`, `larray-eurostat` and `larrayenv` packages switched from `gdementen` to `larray-project` (closes [issue 560](#)).

Fixes

- fixed binary operations between a session and an array object (closes [issue 807](#)).
- fixed `Array.reindex()` printing a spurious warning message when the `axes_to_reindex` argument was the name of the axis to reindex (closes [issue 812](#)).
- fixed `zip_array_values()` and `zip_array_items()` functions not available when importing the entire `larray` library as `from larray import *` (closes [issue 816](#)).
- fixed wrong axes and groups names when loading a session from an HDF file (closes [issue 803](#)).

EDITOR

New features

- added `debug()` function which opens an editor window with an extra widget to navigate back in the call stack (the chain of functions called to reach the current line of code).

Miscellaneous improvements

- Sizes of the main window and the resizable components are saved when closing the viewer and restored when it is reopened (closes [issue 165](#)).
- added keyword arguments `rtol`, `atol` and `nans_equal` to the `compare()` function (closes [issue 172](#)).
- `run_editor_on_exception()` now uses `debug()` so that one can inspect what the state was in all functions traversed to reach the code which triggered the exception.

6.1.2 Version 0.31

Released on 2019-08-09.

- added the *ExcelReport* class allowing to generate multiple graphs in an Excel file at once (closes [issue 676](#)).
- fixed binary operations (+, -, *, etc.) between an LArray and a (scalar) Group which silently gave a wrong result (closes [issue 797](#)).
- fixed taking a subset of an array with boolean labels for an axis if the user explicitly specify the axis (closes [issue 735](#)). When the user does not specify the axis, it currently fails but it is unclear what to do in that case (see [issue 794](#)).
- fixed a regression in 0.30: `X.axis_name[groups]` failed when groups were originally defined on axes with the same name (i.e. when the operation was not actually needed). Closes [issue 787](#).

6.1.3 Version 0.30

Released on 2019-06-27.

- `stack()` axis argument was renamed to `axes` to reflect the fact that the function can now stack along multiple axes at once (see below).
- to accommodate for the “simpler pattern language” now supported for those functions, using a regular expression in `Axis.matching()` or `Group.matching()` now requires passing the pattern as an explicit `regex` keyword argument instead of just the first argument of those methods. For example `my_axis.matching('test.*')` becomes `my_axis.matching(regex='test.*')`.
- `LArray.as_table()` is deprecated because it duplicated functionality found in `LArray.dump()`. Please only use `LArray.dump()` from now on.
- renamed `a_min` and `a_max` arguments of `LArray.clip()` to `minval` and `maxval` respectively and made them optional (closes [issue 747](#)).
- modified the behavior of the `pattern` argument of `Session.filter()` to actually support patterns instead of only checking if the object names start with the pattern. Special characters include `?` for matching any single character and `*` for matching any number of characters. Closes [issue 703](#).

Warning: If you were using `Session.filter`, you must add a `*` to your pattern to keep your code working. For example, `my_session.filter('test')` must be changed to `my_session.filter('test*')`.

- `LArray.equals()` now returns `True` for arrays even when axes are in a different order or some axes are missing on either side (but the data is constant over that axis on the other side). Closes [issue 237](#).

Warning: If you were using `LArray.equals()` **and** want to keep the old, stricter, behavior, you must add `check_axes=True`.

- added `set_options()` and `get_options()` functions to respectively set and get options for larray. Available options currently include `display_precision` for controlling the number of decimal digits used when showing floating point numbers, `display_maxlines` to control the maximum number of lines to use when displaying an array, etc. `set_options()` can be used either like a normal function to set the options globally or within a `with` block to set them only temporarily. Closes [issue 274](#).
- implemented `read_stata()` and `LArray.to_stata()` to read arrays from and write arrays to Stata .dta files.
- implemented `LArray.isin()` method to check whether each value of an array is contained in a list (or array) of values.
- implemented `LArray.unique()` method to compute unique values (or sub-arrays) for an array, optionally along axes.
- implemented `LArray.apply()` method to apply a python function to all values of an array or to all sub-arrays along some axes of an array and return the result. This is an extremely versatile method as it can be used both with aggregating functions or element-wise functions.
- implemented `LArray.apply_map()` method to apply a transformation mapping to array elements. For example, this can be used to transform some numeric codes to labels.
- implemented `LArray.reverse()` method to reverse one or several axes of an array (closes [issue 631](#)).
- implemented `LArray.roll()` method to roll the cells of an array n-times to the right along an axis. This is similar to `LArray.shift()`, except that cells which are pushed “outside of the axis” are reintroduced on the opposite side of the axis instead of being dropped.
- implemented `Axis.apply()` method to transform an axis labels by a function and return a new `Axis`.
- added `Session.update()` method to add and modify items from an existing session by passing either another session or a dict-like object or an iterable object with (key, value) pairs (closes [issue 754](#)).
- implemented `AxisCollection.rename()` to rename axes of an `AxisCollection`, independently of any array.
- implemented `AxisCollection.set_labels()` (closes [issue 782](#)).
- implemented `wrap_elementwise_array_func()` function to make a function defined in another library work with `LArray` arguments instead of with numpy arrays.
- implemented `LArray.keys()`, `LArray.values()` and `LArray.items()` methods to respectively loop on an array labels, values or (key, value) pairs.
- implemented `zip_array_values()` and `zip_array_items()` to loop respectively on several arrays values or (key, value) pairs.
- implemented `AxisCollection.iter_labels()` to iterate over all (possible combinations of) labels of the axes of the collection.

- improved speed of `read_hdf()` function when reading a stored LArray object dumped with the current and future version of larray. To get benefit of the speedup of reading arrays dumped with older versions of larray, please read and re-dump them. Closes [issue 563](#).
- allowed to not specify the axes in `LArray.set_labels()` (closes [issue 634](#)):

```
>>> a = ndtest('nat=BE,FO;sex=M,F')
>>> a
nat\sex  M  F
      BE  0  1
      FO  2  3
>>> a.set_labels({'M': 'Men', 'BE': 'Belgian'})
nat\sex  Men  F
Belgian    0  1
      FO    2  3
```

- `LArray.set_labels()` can now take functions to transform axes labels (closes [issue 536](#)).

```
>>> arr = ndtest((2, 2))
>>> arr
a\b  b0  b1
a0    0  1
a1    2  3
>>> arr.set_labels('a', str.upper)
a\b  b0  b1
A0    0  1
A1    2  3
```

- implemented the same “simpler pattern language” in `Axis.matching()` and `Group.matching()` than in `Session.filter()`, in addition to regular expressions (which now require using the `regexp` argument).
- `stack()` can now stack along several axes at once (closes [issue 56](#)).

```
>>> country = Axis('country=BE,FR,DE')
>>> gender = Axis('gender=M,F')
>>> stack({'BE', 'M'): 0,
...      ('BE', 'F'): 1,
...      ('FR', 'M'): 2,
...      ('FR', 'F'): 3,
...      ('DE', 'M'): 4,
...      ('DE', 'F'): 5},
...      (country, gender))
country\gender  M  F
              BE  0  1
              FR  2  3
              DE  4  5
```

- `stack()` using a dictionary as elements can now use a simple axis name instead of requiring a full axis object. This will print a warning on Python < 3.7 though because the ordering of labels is not guaranteed in that case. Closes [issue 755](#) and [issue 581](#).
- `stack()` using keyword arguments can now use a simple axis name instead of requiring a full axis object, even on Python < 3.6. This will print a warning though because the ordering of labels is not guaranteed in that case.
- added `password` argument to `Workbook.save()` to allow protecting Excel files with a password.
- added option `exact` to `join` argument of `Axis.align()` and `LArray.align()` methods. Instead of aligning, passing `join='exact'` to the `align` method will raise an error when axes are not equal. Closes [issue 338](#).

- made `Axis.by()` and `Group.by()` return a list of named groups instead of anonymous groups. By default, group names are defined as `<start>:<end>`. This can be changed via the new `template` argument:

```
>>> age = Axis('age=0..6')
>>> age
Axis([0, 1, 2, 3, 4, 5, 6], 'age')
>>> age.by(3)
(age.i[0:3] >> '0:2', age.i[3:6] >> '3:5', age.i[6:7] >> '6')
>>> age.by(3, step=2)
(age.i[0:3] >> '0:2', age.i[2:5] >> '2:4', age.i[4:7] >> '4:6', age.i[6:7] >> '6')
>>> age.by(3, template='{start}-{end}')
(age.i[0:3] >> '0-2', age.i[3:6] >> '3-5', age.i[6:7] >> '6')
```

Closes [issue 669](#).

- allowed to specify an axis by its position when selecting a subset of an array using the string notation:

```
>>> pop_mouv = ndtest('geo_from=BE,FR,UK;geo_to=BE,FR,UK')
>>> pop_mouv
geo_from\geo_to  BE  FR  UK
                BE   0   1   2
                FR   3   4   5
                UK   6   7   8
>>> pop_mouv['0[BE, UK]'] # equivalent to pop_mouv[pop_mouv.geo_from['BE,UK']]
geo_from\geo_to  BE  FR  UK
                BE   0   1   2
                UK   6   7   8
>>> pop_mouv['1.i[0, 2]'] # equivalent to pop_mouv[pop_mouv.geo_to.i[0, 2]]
geo_from\geo_to  BE  UK
                BE   0   2
                FR   3   5
                UK   6   8
```

Closes [issue 671](#).

- added documentation and examples for `where()`, `maximum()` and `minimum()` functions (closes [issue 700](#))
- updated the Working With Sessions section of the tutorial (closes [issue 568](#)).
- added `dtype` argument to `LArray` to set the type of the array explicitly instead of relying on auto-detection.
- added `dtype` argument to `stack` to set the type of the resulting array explicitly instead of relying on auto-detection.
- allowed to pass a single axis or group as `axes_to_reindex` argument of the `LArray.reindex()` method (closes [issue 712](#)).
- `LArray.dump()` gained a few extra arguments to further customize output : - `axes_names` : to specify whether or not the output should contain the axes names (and which) - `maxlines` and `edgeitems` : to dump only the start and end of large arrays - `light` : to output axes labels only when they change instead of repeating them on each line - `na_repr` : to specify how to represent N/A (NaN) values
- substantially improved performance of creating, iterating, and doing a few other operations over `larray` objects. This solves a few pathological cases of slow operations, especially those involving many small-ish arrays but sadly the overall performance improvement is negligible over most of the real-world models using `larray` that we tested these changes on.
- fixed dumping to Excel arrays of “object” dtype containing NaN values using numpy float types (fixes the infamous 65535 bug).
- fixed `LArray.divnot0()` being slow when the divisor has many axes and many zeros (closes [issue 705](#)).

- fixed maximum length of sheet names (31 characters instead of 30 characters) when adding a new sheet to an Excel Workbook (closes [issue 713](#)).
- fixed missing documentation of many functions in *Utility Functions* section of the API Reference (closes [issue 698](#)).
- fixed arithmetic operations between two sessions returning a nan value for each axis and group (closes [issue 725](#)).
- fixed dumping sessions with metadata in HDF format (closes [issue 702](#)).
- fixed minimum version of pandas to install. The minimum version is now 0.20.0.
- fixed from_frame for dataframes with non string index names.
- fixed creating an LSet from an IGroup with a (single) scalar key

```
>>> a = Axis('a=a0,a1,a2')
>>> a.i[1].set()
a['a1'].set()
```

6.1.4 Version 0.29

Released on 2018-09-07.

Syntax changes

- deprecated title attribute of LArray objects and title argument of array creation functions. A title is now considered as a metadata and must be added as:

```
>>> # add title at array creation
>>> arr = ndtest((3, 3), meta=[('title', 'array for testing')])
```

```
>>> # or after array creation
>>> arr = ndtest((3, 3))
>>> arr.meta.title = 'array for testing'
```

See below for more information about metadata handling.

- renamed LArray.drop_labels() to LArray.ignore_labels() to avoid confusion with the new LArray.drop() method (closes [issue 672](#)).
- renamed Session.array_equals() to Session.element_equals() because this method now also compares axes and groups in addition to arrays.
- renamed Sheet.load() and Range.load() nb_index argument to nb_axes to be consistent with all other input functions (read_*). Sheet and Range are the objects one gets when taking subsets of the excel Workbook objects obtained via open_excel() (closes [issue 648](#)).
- deprecated the element_equal() function in favor of the LArray.eq() method (closes [issue 630](#)) to be consistent with other future methods for operations between two arrays.
- renamed nan_equals argument of LArray.equals() and LArray.eq() methods to nans_equal because it is grammatically more correct and is explained more naturally as “whether two nans should be considered equal”.
- LArray.insert() pos and axis arguments are deprecated because those were only useful for very specific cases and those can easily be rewritten by using an indices group (axis.i[pos]) for the before argument instead (closes [issue 652](#)).

New features

- allowed arrays to have metadata (e.g. title, description, authors, ...).

Metadata can be added when creating arrays:

```
>>> # for Python <= 3.5
>>> arr = ndtest((3, 3), meta=[('title', 'array for testing'), ('author', 'John_
↳Smith')])
```

```
>>> # for Python >= 3.6
>>> arr = ndtest((3, 3), meta=Metadata(title='array for testing', author='John_
↳Smith'))
```

To access all existing metadata, use `array.meta`, for example:

```
>>> arr.meta
title: array for testing
author: John Smith
```

To access some specific existing metadata, use `array.meta.<name>`, for example:

```
>>> arr.meta.author
'John Smith'
```

Updating some existing metadata, or creating new metadata (the metadata is added if there was no metadata using that name) should be done using `array.meta.<name> = <value>`. For example:

```
>>> arr.meta.city = 'London'
```

To remove some metadata, use `del array.meta.<name>`, for example:

```
>>> del arr.meta.city
```

Note:

- Currently, only the HDF (.h5) file format supports saving and loading array metadata.
- Metadata is not kept when actions or methods are applied on an array except for operations modifying the object in-place, such as `pop[age < 10] = 0`, and when the method `copy()` is called. Do not add metadata to an array if you know you will apply actions or methods on it before dumping it.

Closes [issue 78](#) and [issue 79](#).

- allowed sessions to have metadata. Session metadata is created and accessed **using the same syntax than for arrays** (`session.meta.<name>`), for example to add metadata to a session at creation:

```
>>> # Python <= 3.5
>>> s = Session([('arr1', ndtest(2)), ('arr2', ndtest(3)), meta=[('title', 'my_
↳title'), ('author', 'John Smith')])
```

```
>>> # Python 3.6+
>>> s = Session(arr1=ndtest(2), arr2=ndtest(3), meta=Metadata(title='my title',
↳author='John Smith'))
```

Note:

- Contrary to array metadata, saving and loading session metadata is supported for all current session file formats: Excel, CSV and HDF (.h5)
 - Metadata is not kept when actions or methods are applied on a session except for operations modifying a specific array, such as: $s['arr1'] = 0$. Do not add metadata to a session if you know you will apply actions or methods on it before dumping it.
-

Closes [issue 640](#).

- implemented `LArray.drop()` to return an array without some labels or indices along an axis (closes [issue 506](#)).

```
>>> arr1 = ndtest((2, 4))
>>> arr1
a\b  b0  b1  b2  b3
a0    0   1   2   3
a1    4   5   6   7
>>> a, b = arr1.axes
```

Dropping a single label

```
>>> arr1.drop('b1')
a\b  b0  b2  b3
a0    0   2   3
a1    4   6   7
```

Dropping multiple labels

```
>>> # arr1.drop('b1,b3')
>>> arr1.drop(['b1', 'b3'])
a\b  b0  b2
a0    0   2
a1    4   6
```

Dropping a slice

```
>>> # arr1.drop('b1:b3')
>>> arr1.drop(b['b1':'b3'])
a\b  b0
a0    0
a1    4
```

Dropping labels by position requires to specify the axis

```
>>> # arr1.drop('b.i[1]')
>>> arr1.drop(b.i[1])
a\b  b0  b2  b3
a0    0   2   3
a1    4   6   7
```

- added new module to create arrays with values generated randomly following a few different distributions, or shuffle an existing array along an axis:

```
>>> from larray.random import *
```

Generate integers between two bounds (0 and 10 in this example)

```
>>> randint(0, 10, axes='a=a0..a2')
a  a0  a1  a2
   3   6   2
```

Generate values following a uniform distribution

```
>>> uniform(axes='a=a0..a2')
a              a0              a1              a2
0.33293756929238394  0.5331412592583252  0.6748786766763107
```

Generate values following a normal distribution ($\mu = 1$ and $\sigma = 2$ in this example)

```
>>> normal(1, scale=2, axes='a=a0..a2')
a              a0              a1              a2
-0.9216651561025018  5.119734598931103  4.4467876992838935
```

Randomly shuffle an existing array along one axis

```
>>> arr = ndtest((3, 3))
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
a2   6   7   8
>>> permutation(arr, axis='b')
a\b  b1  b2  b0
a0   1   2   0
a1   4   5   3
a2   7   8   6
```

Generate values by randomly choosing between specified values (5, 10 and 15 in this example), potentially with a specified probability for each value (respectively a 30%, 50%, 20% probability of occurring in this example).

```
>>> choice([5, 10, 15], p=[0.3, 0.5, 0.2], axes='a=a0,a1;b=b0..b2')
a\b  b0  b1  b2
a0   15  10  10
a1   10   5  10
```

Same as above with labels and probabilities given as a one dimensional LArray

```
>>> proba = LArray([0.3, 0.5, 0.2], Axis([5, 10, 15], 'outcome'))
>>> proba
outcome      5      10      15
           0.3  0.5  0.2
>>> choice(p=proba, axes='a=a0,a1;b=b0..b2')
a\b  b0  b1  b2
a0   10  15   5
a1   10   5  10
```

- made a few useful constants accessible directly from the larray module: nan, inf, pi, e and euler_gamma. Like for any Python functionality, you can choose how to import and use them. For example, for pi:

```
>>> from larray import *
>>> pi
3.141592653589793
OR
>>> from larray import pi
```

(continues on next page)

(continued from previous page)

```
>>> pi
3.141592653589793
OR
>>> import larray as la
>>> la.pi
3.141592653589793
```

- added `Group.equals()` method which compares group names, associated axis names and labels between two groups:

```
>>> a = Axis('a=a0..a3')
>>> a02 = a['a0:a2'] >> 'group_a'
>>> # different group name
>>> a02.equals(a['a0:a2'])
False
>>> # different axis name
>>> other_axis = a.rename('other_name')
>>> a02.equals(other_axis['a0:a2'] >> 'group_a')
False
>>> # different labels
>>> a02.equals(a['a1:a3'] >> 'group_a')
False
```

Miscellaneous improvements

- completely rewritten the ‘Load And Dump Arrays, Sessions, Axes And Groups’ section of the tutorial (closes [issue 645](#))
- saving or loading a session from a file now includes *Axis* and *Group* objects in addition to arrays (closes [issue 578](#)).

Create a session containing axes, groups and arrays

```
>>> a, b = Axis("a=a0..a2"), Axis("b=b0..b2")
>>> a01 = a['a0,a1'] >> 'a01'
>>> arr1, arr2 = ndtest((a, b)), ndtest(a)
>>> s = Session([('a', a), ('b', b), ('a01', a01), ('arr1', arr1), ('arr2',
↪arr2)])
```

Saving a session will save axes, groups and arrays

```
>>> s.save('session.h5')
```

Loading a session will load axes, groups and arrays

```
>>> s2 = s.load('session.h5')
>>> s2
Session(arr1, arr2, a, b, a01)
```

Note: All axes and groups of a session are stored in the same CSV file/Excel sheet/HDF group named respectively `__axes__` and `__groups__`.

- vastly improved indexing using arrays (of labels, indices or booleans). Many advanced cases did not work, including when combining several indexing arrays, or when (one of) the indexing array(s) had an axis present

in the array.

First let's create some test axes

```
>>> a, b, c = ndtest((2, 3, 2)).axes
```

Then create a test array.

```
>>> arr = ndtest((a, b))
>>> arr
a\b b0 b1 b2
a0  0  1  2
a1  3  4  5
```

If the key array has an axis not already present in arr (e.g. c), the target axis (a) is replaced by the extra axis (c). This already worked previously.

```
>>> key = LArray(['a1', 'a0'], c)
>>> key
c  c0  c1
   a1  a0
>>> arr[key]
c\b  b0  b1  b2
c0   3  4  5
c1   0  1  2
```

If the key array has the target axis, the axis stays the same, but the data is reordered (this also worked previously):

```
>>> key = LArray(['b1', 'b0', 'b2'], b)
>>> key
b  b0  b1  b2
   b1  b0  b2
>>> arr[key]
a\b  b0  b1  b2
a0   1  0  2
a1   4  3  5
```

From here on, the examples shown did not work previously...

Now, if the key contains another axis present in the array (b) which is not the target axis (a), the target axis completely disappears (both axes are replaced by the key axis):

```
>>> key = LArray(['a0', 'a1', 'a0'], b)
>>> key
b  b0  b1  b2
   a0  a1  a0
>>> arr[key]
b  b0  b1  b2
   0  4  2
```

If the key has both the target axis (a) and another existing axis (b)

```
>>> key
a\b b0 b1 b2
a0 a0 a1 a0
a1 a1 a0 a1
>>> arr[key]
```

(continues on next page)

(continued from previous page)

a\b	b0	b1	b2
a0	0	4	2
a1	3	1	5

If the key has both another existing axis (a) and an extra axis (c)

```
>>> key
a\c  c0  c1
a0   b0  b1
a1   b2  b0
>>> arr[key]
a\c  c0  c1
a0   0   1
a1   5   3
```

It also works if the key has the target axis (a), another existing axis (b) and an extra axis (c), but this is not shown for brevity.

- updated `Session.summary()` so as to display all kinds of objects and allowed to pass a function returning a string representation of an object instead of passing a pre-defined string template (closes [issue 608](#)):

```
>>> axis1 = Axis("a=a0..a2")
>>> group1 = axis1['a0,a1'] >> 'a01'
>>> arr1 = ndtest((2, 2), title='array 1', dtype=np.int64)
>>> arr2 = ndtest(4, title='array 2', dtype=np.int64)
>>> arr3 = ndtest((3, 2), title='array 3', dtype=np.int64)
>>> s = Session([('axis1', axis1), ('group1', group1), ('arr1', arr1), ('arr2',
↪arr2), ('arr3', arr3)])
```

Using the default template

```
>>> print(s.summary())
axis1: a ['a0' 'a1' 'a2'] (3)
group1: a['a0', 'a1'] >> a01 (2)
arr1: a, b (2 x 2) [int64]
      array 1
arr2: a (4) [int64]
      array 2
arr3: a, b (3 x 2) [int64]
      array 3
```

Using a specific template

```
>>> def print_array(key, array):
...     axes_names = ', '.join(array.axes.display_names)
...     shape = ' x '.join(str(i) for i in array.shape)
...     return "{} -> {} ({})\n title = {}\n dtype = {}".format(key, axes_
↪names, shape,
                                                                    array.title,
↪array.dtype)
>>> template = {Axis: "{key} -> {name} [{labels}] ({length})",
...               Group: "{key} -> {name}: {axis_name} {labels} ({length})",
...               LArray: print_array}
>>> print(s.summary(template))
axis1 -> a ['a0' 'a1' 'a2'] (3)
group1 -> a01: a ['a0', 'a1'] (2)
arr1 -> a, b (2 x 2)
```

(continues on next page)

(continued from previous page)

```

title = array 1
dtype = int64
arr2 -> a (4)
title = array 2
dtype = int64
arr3 -> a, b (3 x 2)
title = array 3
dtype = int64

```

- methods `Session.equals()` and `Session.element_equals()` now also compare axes and groups in addition to arrays (closes [issue 610](#)):

```

>>> a = Axis('a=a0..a2')
>>> a01 = a['a0,a1'] >> 'a01'
>>> s1 = Session([('a', a), ('a01', a01), ('arr1', ndtest(2)), ('arr2', ndtest((2,
↪ 2))))])
>>> s2 = Session([('a', a), ('a01', a01), ('arr1', ndtest(2)), ('arr2', ndtest((2,
↪ 2))))])

```

Identical sessions

```

>>> s1.element_equals(s2)
name      a   a01  arr1  arr2
      True  True  True  True

```

Different value(s) between two arrays

```

>>> s2.arr1['a1'] = 0
>>> s1.element_equals(s2)
name      a   a01  arr1  arr2
      True  True  False  True

```

Different label(s)

```

>>> s2.arr2 = ndtest("b=b0,b1; a=a0,a1")
>>> s2.a = Axis('a=a0,a1')
>>> s1.element_equals(s2)
name      a   a01  arr1  arr2
      False  True  False  False

```

Extra/missing objects

```

>>> s2.arr3 = ndtest((3, 3))
>>> del s2.a
>>> s1.element_equals(s2)
name      a   a01  arr1  arr2  arr3
      False  True  False  False  False

```

- added arguments `wide` and `value_name` to methods `LArray.as_table()` and `LArray.dump()` like in `LArray.to_excel()` and `LArray.to_csv()` (closes [issue 653](#)).
- the `from_series()` function supports Pandas series with a MultiIndex (closes [issue 465](#))
- the `stack()` function supports any array-like object instead of only LArray objects.

```

>>> stack(a0=[1, 2, 3], a1=[4, 5, 6], axis='a')
{0}*\a  a0  a1

```

(continues on next page)

(continued from previous page)

0	1	4
1	2	5
2	3	6

- made some operations on Excel Workbooks a bit faster by telling Excel to avoid updating the screen when the Excel instance is not visible anyway. This affects all workbooks opened via `open_excel()` as well as `read_excel()` and `LArray.to_excel()` when using the default xlwings engine.
- made the documentation link in Windows start menu version-specific (instead of always pointing to the latest release) so that users do not inadvertently use the latest release syntax when using an older version of larray (closes [issue 142](#)).
- added menu bar with undo/redo when editing single arrays (as a byproduct of [issue 133](#)).

Fixes

- fixed Copy(to Excel)/Paste/Plot in the editor not working for 1D and 2D arrays (closes [issue 140](#)).
- fixed Excel add-ins not loaded when opening an Excel Workbook by calling the `LArray.to_excel()` method with no path or via “Copy to Excel (CTRL+E)” in the editor (closes [issue 154](#)).
- made LArray support Pandas versions ≥ 0.21 (closes [issue 569](#))
- fixed current active Excel Workbook being closed when calling the `LArray.to_excel()` method on an array with `-1` as `filepath` argument (closes [issue 473](#)).
- fixed `LArray.split_axes()` when splitting a single axis and using the `names` argument (e.g. `arr.split_axes('bd', names=('b', 'd'))`).
- fixed splitting an anonymous axis without specifying the `names` argument.

```
>>> combined = ndtest('a0_b0,a0_b1,a0_b2,a1_b0,a1_b1,a1_b2')
>>> combined
{0}  a0_b0  a0_b1  a0_b2  a1_b0  a1_b1  a1_b2
      0      1      2      3      4      5
>>> combined.split_axes(0)
{0}\{1}  b0  b1  b2
        a0  0  1  2
        a1  3  4  5
```

- fixed `LArray.combine_axes()` with `wildcard=True`.
- fixed taking a subset of an array by giving an index along a specific axis using a string (strings like `"axisname.i[pos]"`).
- fixed the editor not working with Python 2 or recent Qt4 versions.

6.1.5 Version 0.28

Released on 2018-03-15.

Backward incompatible changes

- changed behavior of operators `session1 == session2` and `session1 != session2`: returns a session of boolean arrays (closes [issue 516](#)):

```
>>> s1 = Session([('arr1', ndtest(2)), ('arr2', ndtest((2, 2)))])
>>> s2 = Session([('arr1', ndtest(2)), ('arr2', ndtest((2, 2)))])
>>> (s1 == s2).arr1
a   a0   a1
    True  True
>>> s2.arr1['a1'] = 0
>>> (s1 == s2).arr1
a   a0   a1
    True False
>>> (s1 != s2).arr1
a   a0   a1
    False True
```

New features

- made it possible to run the tutorial online (as a Jupyter notebook) by clicking on the `launch|binder` badge on top of the tutorial web page (closes [issue 73](#))
- added methods `array_equals` and `equals` to `Session` object to compare arrays from two sessions. The method `array_equals` return a boolean value for each array while the method `equals` returns a unique boolean value (True if all arrays of both sessions are equal, False otherwise):

```
>>> s1 = Session([('arr1', ndtest(2)), ('arr2', ndtest((2, 2)))])
>>> s2 = Session([('arr1', ndtest(2)), ('arr2', ndtest((2, 2)))])
>>> s1.array_equals(s2)
name  arr1  arr2
      True  True
>>> s1.equals(s2)
True
```

Different value(s)

```
>>> s2.arr1['a1'] = 0
>>> s1.array_equals(s2)
name  arr1  arr2
      False True
>>> s1.equals(s2)
False
```

Different label(s)

```
>>> from larray import ndrange
>>> s2.arr2 = ndrange("b=b0,b1; a=a0,a1")
>>> s1.array_equals(s2)
name  arr1  arr2
      False False
>>> s1.equals(s2)
False
```

Extra/missing array(s)

```
>>> s2.arr3 = ndtest((3, 3))
>>> s1.array_equals(s2)
name   arr1   arr2   arr3
      False False False
>>> s1.equals(s2)
False
```

Closes [issue 517](#).

- added method *equals* to *LArray* object to compare two arrays:

```
>>> arr1 = ndtest((2, 3))
>>> arr1
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> arr2 = arr1.copy()
>>> arr1.equals(arr2)
True
>>> arr2['b1'] += 1
>>> arr1.equals(arr2)
False
>>> arr3 = arr1.set_labels('a', ['x0', 'x1'])
>>> arr1.equals(arr3)
False
```

Arrays with nan values

```
>>> arr1 = ndtest((2, 3), dtype=float)
>>> arr1['a1', 'b1'] = nan
>>> arr1
a\b  b0  b1  b2
a0  0.0  1.0  2.0
a1  3.0 nan  5.0
>>> arr2 = arr1.copy()
>>> # By default, an array containing nan values is never equal to_
    ↪ another array,
>>> # even if that other array also contains nan values at the same_
    ↪ positions.
>>> # The reason is that a nan value is different from *anything*,_
    ↪ including itself.
>>> arr1.equals(arr2)
False
>>> # set flag nan_equal to True to override this behavior
>>> arr1.equals(arr2, nan_equal=True)
True
```

This method also includes the arguments *rtol* (relative tolerance) and *atol* (absolute tolerance) allowing to test the equality between two arrays within a given relative or absolute tolerance:

```
>>> arr1 = LArray([6., 8.], "a=a0,a1")
>>> arr1
a   a0   a1
   6.0  8.0
>>> arr2 = LArray([5.999, 8.001], "a=a0,a1")
>>> arr2
a   a0   a1
   5.999 8.001
```

(continues on next page)

(continued from previous page)

```

>>> arr1.equals(arr2)
False
>>> # equals returns True if abs(array1 - array2) <= (atol + rtol *
↳abs(array2))
>>> arr1.equals(arr2, atol=0.01)
True
>>> arr1.equals(arr2, rtol=0.01)
True

```

Closes [issue 488](#) and [issue 518](#).

- added *Load from Script* in the File menu of the editor allowing to load commands from an existing Python file (closes [issue 96](#)).
- added *Edit* menu allowing to undo and redo changes of array values by editing cells and removed *Apply* and *Discard* buttons. Changes are now kept when switching from an array to another instead of losing them as previously (closes [issue 32](#)).
- allowed to provide an absolute or relative tolerance value when comparing arrays through the *compare* function (closes [issue 131](#)).
- made the editor able to detect and display plot objects stored in tuple, list or arrays. For example, arrays of plot objects are returned when using *subplots=True* option in calls of *plot* method:

```

>>> a = ndtest('sex=M,F; nat=BE,FO; year=2000..2017')
>>> # display 4 plots vertically placed (one plot for each pair (sex,
↳nationality))
>>> a.plot(subplots=True)
>>> # display 4 plots ordered in a 2 x 2 grid
>>> a.plot(subplots=True, layout=(2, 2))

```

Closes [issue 135](#).

Miscellaneous improvements

- functions *local_arrays*, *global_arrays* and *arrays* returns a session excluding arrays starting by an underscore by default. To include them, set the flag *include_private* to True (closes [issue 513](#)):

```

>>> global_arr1 = ndtest((2, 2))
>>> _global_arr2 = ndtest((3, 3))
>>> def foo():
...     local_arr1 = ndtest(2)
...     _local_arr2 = ndtest(3)
...
...     # exclude arrays starting with '_' by default
...     s = arrays()
...     print(s.names)
...
...     # use flag 'include_private' to include arrays starting with '_'
...     s = arrays(include_private=True)
...     print(s.names)
>>> foo()
['global_arr1', 'local_arr1']
['_global_arr2', '_local_arr2', 'global_arr1', 'local_arr1']

```

- implemented sessions binary operations with non sessions objects (closes [issue 514](#) and [issue 515](#)):

```
>>> s = Session(arr1=ndtest((2, 2)), arr2=ndtest((3, 3)))
>>> s.arr1
a\b  b0  b1
a0    0   1
a1    2   3
>>> s.arr2
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
a2    6   7   8
```

Add a scalar to all arrays

```
>>> # equivalent to s2 = 3 + s
>>> s2 = s + 3
>>> s2.arr1
a\b  b0  b1
a0    3   4
a1    5   6
>>> s2.arr2
a\b  b0  b1  b2
a0    3   4   5
a1    6   7   8
a2    9  10  11
```

Apply binary operations between two sessions

```
>>> sdiff = (s2 - s) / s
>>> sdiff.arr1
a\b  b0  b1
a0  inf  3.0
a1  1.5  1.0
>>> sdiff.arr2
a\b  b0  b1  b2
a0  inf  3.0  1.5
a1  1.0  0.75 0.6
a2  0.5  0.43 0.375
```

- added possibility to call the method *reindex* with a group (closes [issue 531](#)):

```
>>> arr = ndtest((2, 2))
>>> arr
a\b  b0  b1
a0    0   1
a1    2   3
>>> b = Axis("b=b2..b0")
>>> arr.reindex('b', b['b1':])
a\b  b1  b0
a0    1   0
a1    3   2
```

- added possibility to call the methods *diff* and *growth_rate* with a group (closes [issue 532](#)):

```
>>> data = [[2, 4, 5, 4, 6], [4, 6, 3, 6, 9]]
>>> a = LArray(data, "sex=M,F; year=2016..2020")
>>> a
sex\year  2016  2017  2018  2019  2020
```

(continues on next page)

(continued from previous page)

```

      M      2      4      5      4      6
      F      4      6      3      6      9
>>> a.diff(a.year[2017:])
sex\year  2018  2019  2020
      M      1     -1      2
      F     -3      3      3
>>> a.growth_rate(a.year[2017:])
sex\year  2018  2019  2020
      M  0.25  -0.2   0.5
      F -0.5   1.0   0.5

```

- function `ndrange` has been deprecated in favor of `sequence` or `ndtest`. Also, an Axis or a list/tuple/collection of axes can be passed to the `ndtest` function (closes [issue 534](#)):

```

>>> ndtest("nat=BE,FO;sex=M,F")
nat\sex  M  F
      BE  0  1
      FO  2  3

```

- allowed to pass a group for argument `axis` of `stack` function (closes [issue 535](#)):

```

>>> b = Axis('b=b0..b2')
>>> stack(b0=ndtest(2), b1=ndtest(2), axis=b[:'b1'])
a\b  b0  b1
a0   0   0
a1   1   1

```

- renamed argument `nb_index` of `read_csv`, `read_excel`, `read_sas`, `from_lists` and `from_string` functions as `nb_axes`. The relation between `nb_index` and `nb_axes` is given by `nb_axes = nb_index + 1`:

For a given file 'arr.csv' with content

```

a,b\c,c0,c1
a0,b0,0,1
a0,b1,2,3
a1,b0,4,5
a1,b1,6,7

```

previous code to read this array such as :

```

>>> # deprecated
>>> arr = read_csv('arr.csv', nb_index=2)

```

must be updated as follow :

```

>>> arr = read_csv('arr.csv', nb_axes=3)

```

Closes [issue 548](#).

- deprecated `nan_equal` function in favor of `element_equal` function. The `element_equal` function has the same optional arguments as the `LArray.equals` method but compares two arrays element-wise and returns an array of booleans:

```

>>> arr1 = LArray([6., np.nan, 8.], "a=a0..a2")
>>> arr1
a  a0  a1  a2
   6.0 nan 8.0

```

(continues on next page)

(continued from previous page)

```

>>> arr2 = LArray([5.999, np.nan, 8.001], "a=a0..a2")
>>> arr2
a      a0      a1      a2
      5.999  nan   8.001
>>> element_equal(arr1, arr2)
a      a0      a1      a2
      False False False
>>> element_equal(arr1, arr2, nan_equals=True)
a      a0      a1      a2
      False True  False
>>> element_equal(arr1, arr2, atol=0.01, nan_equals=True)
a      a0      a1      a2
      True  True  True
>>> element_equal(arr1, arr2, rtol=0.01, nan_equals=True)
a      a0      a1      a2
      True  True  True

```

Closes [issue 593](#).

- renamed argument *transpose* by *wide* in *to_csv* method.
- added argument *wide* in *to_excel* method. When argument *wide* is set to False, the array is exported in “narrow” format, i.e. one column per axis plus one value column:

```

>>> arr = ndtest((2, 3))
>>> arr
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5

```

Default behavior (*wide=True*):

```

>>> arr.to_excel('my_file.xlsx')
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5

```

With *wide=False*:

```

>>> arr.to_excel('my_file.xlsx', wide=False)
a  b  value
a0 b0    0
a0 b1    1
a0 b2    2
a1 b0    3
a1 b1    4
a1 b2    5

```

Argument *transpose* has a different purpose than *wide* and is mainly useful to allow multiple axes as header when exporting arrays with more than 2 dimensions. Closes [issue 575](#) and [issue 371](#).

- added argument *wide* to *read_csv* and *read_excel* functions. If False, the array to be loaded is assumed to be stored in “narrow” format:

```

>>> # assuming the array was saved using command: arr.to_excel('my_file.xlsx',
↳ wide=False)
>>> read_excel('my_file.xlsx', wide=False)

```

(continues on next page)

(continued from previous page)

```
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
```

Closes [issue 574](#).

- added argument *name* to *to_series* method allowing to set a name to the Pandas Series returned by the method.
- added argument *value_name* to *to_csv* and *to_excel* allowing to change the default name ('value') to the column containing the values when the argument *wide* is set to False:

```
>>> arr.to_csv('my_file.csv', wide=False, value_name='data')
a,b,data
a0,b0,0
a0,b1,1
a0,b2,2
a1,b0,3
a1,b1,4
a1,b2,5
```

Closes [issue 549](#).

- renamed argument *sheetname* of *read_excel* function as *sheet* (closes [issue 587](#)).
- Renamed *sheet_name* of *LArray.to_excel* to *sheet* since it can also be an index (closes [issue 580](#)).
- allowed to create axes with zero padded string labels (closes [issue 533](#)):

```
>>> Axis('zero_padding=01,02,03,10,11,12')
Axis(['01', '02', '03', '10', '11', '12'], 'zero_padding')
```

- added a dropdown menu containing recently used files in dialog boxes of *Save Command History To Script* and *Load from Script* from File menu.

Fixes

- fixed passing a scalar group from an external axis to get a subset of an array (closes [issue 178](#)):

```
>>> arr = ndtest((3, 2))
>>> arr['a1']
b  b0  b1
   2   3
>>> alt_a = Axis("alt_a=a1..a2")
>>> arr[alt_a['a1']]
b  b0  b1
   2   3
>>> arr[alt_a.i[0]]
b  b0  b1
   2   3
```

- fixed subscript a string LGroup key (closes [issue 437](#)):

```
>>> axis = Axis("a=a0,a1")
>>> axis['a0'][0]
'a'
```

- fixed *Axis.union*, *Axis.intersection* and *Axis.difference* when passed value is a single string (closes [issue 489](#)):

```
>>> a = Axis('a=a0..a2')
>>> a.union('a1')
Axis(['a0', 'a1', 'a2'], 'a')
>>> a.union('a3')
Axis(['a0', 'a1', 'a2', 'a3'], 'a')
>>> a.union('a1..a3')
Axis(['a0', 'a1', 'a2', 'a3'], 'a')
>>> a.intersection('a1..a3')
Axis(['a1', 'a2'], 'a')
>>> a.difference('a1..a3')
Axis(['a0'], 'a')
```

- fixed `to_excel` applied on ≥ 2 D arrays using `transpose=True` (closes [issue 579](#))

```
>>> arr = ndtest((2, 3))
>>> arr.to_excel('my_file.xlsx', transpose=True)
b\a  a0  a1
b0    0   3
b1    1   4
b2    2   5
```

- fixed aggregation on arrays containing zero padded string labels (closes [issue 522](#)):

```
>>> arr = ndtest('zero_padding=01,02,03,10,11,12')
>>> arr
zero_padding  01  02  03  10  11  12
              0   1   2   3   4   5
>>> arr.sum('01,02,03 >> 01_03; 10')
zero_padding  01_03  10
              3     3
```

6.1.6 Version 0.27

Released on 2017-11-30.

Syntax changes

- renamed `Axis.translate` to `Axis.index` (closes [issue 479](#)).
- deprecated `reverse` argument of `sort_values` and `sort_axes` methods in favor of `ascending` argument (defaults to `True`). Closes [issue 540](#).

Backward incompatible changes

- labels are checked during array subset assignment (closes [issue 269](#)):

```
>>> arr = ndtest(4)
>>> arr
a  a0  a1  a2  a3
   0   1   2   3
>>> arr['a0,a1'] = arr['a2,a3']
ValueError: incompatible axes:
Axis(['a0', 'a1'], 'a')
```

(continues on next page)

(continued from previous page)

```
vs
Axis(['a2', 'a3'], 'a')
```

previous behavior can be recovered through `drop_labels` or by changing labels via `set_labels` or `set_axes`:

```
>>> arr['a0,a1'] = arr['a2,a3'].drop_labels('a')
>>> arr['a0,a1'] = arr['a2,a3'].set_labels('a', {'a2': 'a0', 'a3': 'a1'})
```

- `from_frame parse_header` argument defaults to *False* instead of *True*.

New features

- implemented `Axis.insert` and `LArray.insert` to add values at a given position of an axis (closes [issue 54](#)).

```
>>> arr1 = ndtest((2, 3))
>>> arr1
a\\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
>>> arr1.insert(42, before='b1', label='b0.5')
a\\b  b0  b0.5 b1  b2
a0    0   42   1   2
a1    3   42   4   5
```

insert an array

```
>>> arr2 = ndtest(2)
>>> arr2
a  a0  a1
   0   1
>>> arr1.insert(arr2, after='b0', label='b0.5')
a\\b  b0  b0.5 b1  b2
a0    0    0   1   2
a1    3    1   4   5
```

insert an array which already has the axis

```
>>> arr3 = ndrange('a=a0,a1;b=b0.1,b0.2') + 42
>>> arr3
a\\b  b0.1  b0.2
a0    42    43
a1    44    45
>>> arr1.insert(arr3, before='b1')
a\\b  b0  b0.1  b0.2  b1  b2
a0    0    42    43   1   2
a1    3    44    45   4   5
```

- added new items in the Help menu of the editor:
 - *Report Issue...*: to report an issue on the Github project website.
 - *Users Discussion...*: redirect to the LArray Users Google Group (you need to be registered to participate).
 - *New Releases And Announces Mailing List...*: redirect to the LArray Announce mailing list.
 - *About*: give information about the editor and the versions of packages currently installed on your computer (closes [issue 88](#)).

- added *Save Command History To Script* in the File menu of the editor allowing to save executed commands in a new or existing Python file.
- added possibility to show only rows with differences when comparing arrays or sessions through the *compare* function in the editor (closes [issue 102](#)).
- added *ascending* argument to methods *indicesofsorted* and *labelsofsorted*. Values are sorted in ascending order by default. Set to *False* to sort values in descending order:

```
>>> arr = LArray([[1, 5], [3, 2], [0, 4]], "nat=BE,FR,IT; sex=M,F")
>>> arr
nat\sex  M  F
      BE  1  5
      FR  3  2
      IT  0  4
>>> arr.indicesofsorted("nat", ascending=False)
nat\sex  M  F
      0  1  0
      1  0  2
      2  2  1
>>> arr.labelsofsorted("nat", ascending=False)
nat\sex  M  F
      0  FR  BE
      1  BE  IT
      2  IT  FR
```

Closes [issue 490](#).

Miscellaneous improvements

- allowed to sort values of an array along an axis (closes [issue 225](#)):

```
>>> a = LArray([[10, 2, 4], [3, 7, 1]], "sex=M,F; nat=EU,FO,BE")
>>> a
sex\nat  EU  FO  BE
      M  10  2  4
      F   3  7  1
>>> a.sort_values(axis='sex')
sex*\nat  EU  FO  BE
      0   3  2  1
      1  10  7  4
>>> a.sort_values(axis='nat')
sex\nat*  0  1  2
      M  2  4  10
      F  1  3  7
```

- method *LArray.sort_values* can be called without argument (closes [issue 478](#)):

```
>>> arr = LArray([0, 1, 6, 3, -1], "a=a0..a4")
>>> arr
a  a0  a1  a2  a3  a4
   0   1   6   3  -1
>>> arr.sort_values()
a  a4  a0  a1  a3  a2
   -1   0   1   3   6
```

If the array has more than one dimension, axes are combined together:

```
>>> a = LArray([[10, 2, 4], [3, 7, 1]], "sex=M,F; nat=EU,FO,BE")
>>> a
sex\nat  EU  FO  BE
      M  10   2   4
      F   3   7   1
>>> a.sort_values()
sex_nat  F_BE  M_FO  F_EU  M_BE  F_FO  M_EU
          1     2     3     4     7    10
```

- when appending/prepending/extending an array, both the original array and the added values will be converted to a data type which can hold both without loss of information. It used to convert the added values to the type of the original array. For example, given an array of integers like:

```
>>> arr = ndtest(3)
a  a0  a1  a2
   0   1   2
```

Trying to add a floating point number to that array used to result in:

```
>>> arr.append('a', 2.5, 'a3')
a  a0  a1  a2  a3
   0   1   2   2
```

Now it will result in:

```
>>> arr.append('a', 2.5, 'a3')
a  a0  a1  a2  a3
 0.0 1.0 2.0 2.5
```

- made the editor more responsive when switching to or changing the filter of large arrays (closes [issue 93](#)).
- added support for coloring numeric values for object arrays (e.g. arrays containing both strings and numbers).
- documentation links in the Help menu of the editor point to the version of the documentation corresponding to the installed version of larray (closes [issue 105](#)).

Fixes

- fixed array values being editable in view() (instead of only in edit()).

6.1.7 Version 0.26.1

Released on 2017-10-25.

Miscellaneous improvements

- Made handling Excel sheets with many blank columns/rows after the data much faster (but still slower than sheets without such blank cells).

Fixes

- fixed reading from and writing to Excel sheets with 16384 columns or 1048576 rows (Excel's maximum).
- fixed LArray.split_axes using a custom separator and not using sort=True or when the split labels are ambiguous with labels from other axes (closes [issue 485](#)).
- fixed reading 1D arrays with non-string labels (closes [issue 495](#)).
- fixed read_csv(sort_columns=True) for 1D arrays (closes [issue 497](#)).

6.1.8 Version 0.26

Released on 2017-10-13.

Syntax changes

- renamed special variable *x* to *X* to let users define an *x* variable in their code without breaking all subsequent code using that special variable (closes [issue 167](#)).
- renamed Axis.startswith, endswith and matches to startingwith, endingwith and matching to avoid a possible confusion with str.startswith and endswith which return booleans (closes [issue 432](#)).
- renamed *na* argument of *read_csv*, *read_excel*, *read_hdf* and *read_sas* functions to *fill_value* to avoid confusion as to what the argument does and to be consistent with *reindex* and *align* (closes [issue 394](#)).
- renamed *split_axis* to *split_axes* to reflect the fact that it can now split several axes at once (see below).
- renamed *sort_axis* to *sort_axes* to reflect the fact that it can sort multiple axes at once (and does so by default).
- renamed several methods with more explicit names (closes [issue 50](#)):
 - *argmax*, *argmin*, *argsort* to *labelofmax*, *labelofmin*, *labelsofsorted*
 - *posargmax*, *posargmin*, *posargsort* to *indexofmax*, *indexofmin*, *indicesofsorted*
- renamed PGroup to IGroup to be consistent with other methods, especially the *.i* methods on axes and arrays (I is for Index – P was for Position).

Backward incompatible changes

- getting a subset using a boolean selection returns an array with labels combined with underscore by defaults (for consistency with *split_axes* and *combine_axes*). Closes [issue 376](#):

```
>>> arr = ndtest((2, 2))
>>> arr
a\b  b0  b1
a0   0   1
a1   2   3
>>> arr[arr < 3]
a_b  a0_b0  a0_b1  a1_b0
      0      1      2
```

New features

- added `global_arrays()` and `arrays()` functions to complement the `local_arrays()` function. They return a `Session` containing respectively all arrays defined in global variables and all available arrays (whether they are defined in local or global variables).

When used outside of a function, these three functions should have the same results, but inside a function `local_arrays()` will return only arrays local to the function, `global_arrays()` will return only arrays defined globally and `arrays()` will return arrays defined either locally or globally. Closes [issue 416](#).

- a `*` symbol is appended to the window title when unsaved changes are detected in the viewer (closes [issue 21](#)).
- implemented `Axis.containing` to create a `Group` with all labels of an axis containing some substring (closes [issue 402](#)).

```
>>> people = Axis(['Bruce Wayne', 'Bruce Willis', 'Arthur Dent'], 'people')
>>> people.containing('Will')
people['Bruce Willis']
```

- implemented `Group.containing`, `startswith`, `endingwith` and `matching` to create a group with all labels of a group matching some criterion (closes [issue 108](#)).

```
>>> group = people.startswith('Bru')
>>> group
people['Bruce Wayne', 'Bruce Willis']
>>> group.containing('Will')
people['Bruce Willis']
```

- implemented `nan_equal()` function to create an array of booleans telling whether each cell of the first array is equal to the corresponding cell in the other array, even in the presence of NaN.

```
>>> arr1 = ndtest(3, dtype=float)
>>> arr1['a1'] = nan
>>> arr1
a   a0   a1   a2
    0.0 nan  2.0
>>> arr2 = arr1.copy()
>>> arr1 == arr2
a   a0   a1   a2
    True False True
>>> nan_equal(arr1, arr2)
a   a0   a1   a2
    True True  True
```

- implemented `from_frame()` to convert a Pandas `DataFrame` to an array:

```
>>> df = ndtest((2, 2, 2)).to_frame()
>>> df
c      c0  c1
a  b
a0 b0   0   1
   b1   2   3
a1 b0   4   5
   b1   6   7
>>> from_frame(df)
a  b\\c  c0  c1
a0  b0   0   1
a0  b1   2   3
```

(continues on next page)

(continued from previous page)

a1	b0	4	5
a1	b1	6	7

- implemented `Axis.split` to split an axis into several.

```
>>> a_b = Axis('a_b=a0_b0,a0_b1,a0_b2,a1_b0,a1_b1,a1_b2')
>>> a_b.split()
[Axis(['a0', 'a1'], 'a'), Axis(['b0', 'b1', 'b2'], 'b')]
```

- added the possibility to load the example dataset used in the tutorial via the menu `File > Load Example` in the viewer

Miscellaneous improvements

- `view()` and `edit()` without argument now display global arrays in addition to local ones (closes [issue 54](#)).
- using the mouse scrollwheel on filter combo boxes will switch to the previous/next label.
- implemented a combobox to choose which color gradient to use and provide a few gradients.
- inverted background colors in the viewer (red for low values and blue for high values). Closes [issue 18](#).
- allowed to pass an array of labels as `new_axis` argument to `reindex` method (closes [issue 384](#)):

```
>>> arr = ndrange('a=v0..v1;b=v0..v2')
>>> arr
a\b  v0  v1  v2
v0   0   1   2
v1   3   4   5
>>> arr.reindex('a', arr.b.labels)
a\b  v0  v1  v2
v0   0   1   2
v1   3   4   5
v2  nan nan nan
```

- allowed to call the `reindex` method using a differently named axis for labels (closes [issue 386](#)):

```
>>> arr = ndrange('a=v0..v1;b=v0..v2')
>>> arr
a\b  v0  v1  v2
v0   0   1   2
v1   3   4   5
>>> arr.reindex('a', arr.b)
a\b  v0  v1  v2
v0   0   1   2
v1   3   4   5
v2  nan nan nan
```

- arguments `fill_value`, `sort_rows` and `sort_columns` of `read_excel` function are also supported by the default `xlwings` engine (closes [issue 393](#)).
- allowed to pass a label or group as `sheet_name` argument of the method `to_excel` or to a Workbook (`open_excel`). Same for `key` argument of the method `to_hdf`. Closes [issue 328](#).

```
>>> arr = ndtest((4, 4, 4))
```



```
>>> # iterate over labels of a given axis
>>> with open_excel('my_file.xlsx') as wb:
>>>     for label in arr.a:
...         wb[label] = arr[label].dump()
...     wb.save()
>>> for label in arr.a:
...     arr[label].to_hdf('my_file.h5', label)
```

```
>>> # create and use a group
>>> even = arr.a['a0,a2'] >> 'even'
>>> arr[even].to_excel('my_file.xlsx', even)
>>> arr[even].to_hdf('my_file.h5', even)
```

```
>>> # special characters : \ / ? * [ or ] in labels or groups are replaced by an _
↳ when exporting to excel
>>> # sheet names cannot exceed 31 characters
>>> g = arr.a['a1,a3,a4'] >> '?name:with*special\[char]'
>>> arr[g].to_excel('my_file.xlsx', g)
>>> print(open_excel('my_file.xlsx').sheet_names())
['_name_with_special__char_']
>>> # special characters \ or / in labels or groups are replaced by an _ when
↳ exporting to HDF file
```

- allowed to pass a Group to `read_excel/read_hdf` as `sheetname/key` argument (closes [issue 439](#)).

```
>>> a, b, c = arr.a, arr.b, arr.c
```

```
>>> # For Excel
>>> new_from_excel = zeros((a, b, c), dtype=int)
>>> for label in a:
...     new_from_excel[label] = read_excel('my_file.xlsx', label)
>>> # But, to avoid loading the file in Excel repeatedly (which is very
↳ inefficient),
>>> # this particular example should rather be written like this:
>>> new_from_excel = zeros((a, b, c), dtype=int)
>>> with open_excel('my_file.xlsx') as wb:
...     for label in a:
...         new_from_excel[label] = wb[label].load()
```

```
>>> # For HDF
>>> new_from_hdf = zeros((a, b, c), dtype=int)
>>> for label in a:
...     new_from_hdf[label] = read_hdf('my_file.h5', label)
```

- allowed setting the name of a Group using another Group or Axis (closes [issue 341](#)):

```
>>> arr = ndrange('axis=a,a0..a3,b,b0..b3,c,c0..c3')
>>> arr
axis  a  a0  a1  a2  a3  b  b0  b1  b2  b3  c  c0  c1  c2  c3
      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
>>> # matches('^.$') will select labels with only one character: 'a', 'b' and 'c'
>>> groups = tuple(arr.axis.startswith(code) >> code for code in arr.axis.matches(
↳ '^.$'))
>>> groups
(axis['a', 'a0', 'a1', 'a2', 'a3'] >> 'a',
 axis['b', 'b0', 'b1', 'b2', 'b3'] >> 'b',
```

(continues on next page)

(continued from previous page)

```
axis['c', 'c0', 'c1', 'c2', 'c3'] >> 'c')
>>> arr.sum(groups)
axis    a    b    c
      10   35   60
```

- allowed to test if an array contains a label using the *in* operator (closes [issue 343](#)):

```
>>> arr = ndrange('age=0..99;sex=M,F')
>>> 'M' in arr
True
>>> 'Male' in arr
False
>>> # this can be useful for example in an 'if' statement
>>> if 102 not in arr:
...     # with 'reindex', we extend 'age' axis to 102
...     arr = arr.reindex('age', Axis('age=0..102'), fill_value=0)
>>> arr.info
103 x 2
age [103]: 0 1 2 ... 100 101 102
sex [2]: 'M' 'F'
```

- allowed to create a group on an axis using labels of another axis (closes [issue 362](#)):

```
>>> year = Axis('year=2000..2017')
>>> even_year = Axis(range(2000, 2017, 2), 'even_year')
>>> group_even_year = year[even_year]
>>> group_even_year
year[2000, 2002, 2004, 2006, 2008, 2010, 2012, 2014, 2016]
```

- *split_axes* (formerly *split_axis*) now allows to split several axes at once (closes [issue 366](#)):

```
>>> combined = ndrange('a_b = a0_b0..a1_b1; c_d = c0_d0..c1_d1')
>>> combined
a_b\c_d  c0_d0  c0_d1  c1_d0  c1_d1
a0_b0      0      1      2      3
a0_b1      4      5      6      7
a1_b0      8      9     10     11
a1_b1     12     13     14     15
>>> combined.split_axes(['a_b', 'c_d'])
a  b  c\d  d0  d1
a0 b0  c0   0   1
a0 b0  c1   2   3
a0 b1  c0   4   5
a0 b1  c1   6   7
a1 b0  c0   8   9
a1 b0  c1  10  11
a1 b1  c0  12  13
a1 b1  c1  14  15
>>> combined.split_axes({'a_b': ('A', 'B'), 'c_d': ('C', 'D')})
A  B  C\D  d0  d1
a0 b0  c0   0   1
a0 b0  c1   2   3
a0 b1  c0   4   5
a0 b1  c1   6   7
a1 b0  c0   8   9
a1 b0  c1  10  11
```

(continues on next page)

(continued from previous page)

a1	b1	c0	12	13
a1	b1	c1	14	15

- argument *axes* of *split_axes* has become optional: defaults to all axes whose name contains the specified delimiter (closes [issue 365](#)):

```
>>> combined = ndrange('a_b = a0_b0..a1_b1; c_d = c0_d0..c1_d1')
>>> combined
a_b\c_d  c0_d0  c0_d1  c1_d0  c1_d1
a0_b0      0      1      2      3
a0_b1      4      5      6      7
a1_b0      8      9     10     11
a1_b1     12     13     14     15
>>> combined.split_axes()
a  b  c\d  d0  d1
a0 b0 c0  0  1
a0 b0 c1  2  3
a0 b1 c0  4  5
a0 b1 c1  6  7
a1 b0 c0  8  9
a1 b0 c1 10 11
a1 b1 c0 12 13
a1 b1 c1 14 15
```

- allowed to perform several axes combinations at once with the *combine_axes()* method (closes [issue 382](#)):

```
>>> arr = ndtest((2, 2, 2, 2))
>>> arr
a  b  c\d  d0  d1
a0 b0 c0  0  1
a0 b0 c1  2  3
a0 b1 c0  4  5
a0 b1 c1  6  7
a1 b0 c0  8  9
a1 b0 c1 10 11
a1 b1 c0 12 13
a1 b1 c1 14 15
>>> arr.combine_axes([('a', 'c'), ('b', 'd')])
a_c\b_d  b0_d0  b0_d1  b1_d0  b1_d1
a0_c0      0      1      4      5
a0_c1      2      3      6      7
a1_c0      8      9     12     13
a1_c1     10     11     14     15
>>> # set output axes names by passing a dictionary
>>> arr.combine_axes({'a', 'c': 'ac', 'b', 'd': 'bd'})
ac\bd  b0_d0  b0_d1  b1_d0  b1_d1
a0_c0      0      1      4      5
a0_c1      2      3      6      7
a1_c0      8      9     12     13
a1_c1     10     11     14     15
```

- allowed to use keyword arguments in *set_labels* (closes [issue 383](#)):

```
>>> a = ndrange('nat=BE,FO;sex=M,F')
>>> a
nat\sex  M  F
```

(continues on next page)

(continued from previous page)

```

    BE  0  1
    FO  2  3
>>> a.set_labels(sex='Men,Women', nat='Belgian,Foreigner')
    nat\sex  Men  Women
    Belgian    0    1
    Foreigner  2    3

```

- allowed passing an axis to `set_labels` as ‘labels’ argument (closes [issue 408](#)).
- added data type (dtype) to `array.info` (closes [issue 454](#)):

```

>>> arr = ndtest((2, 2), dtype=float)
>>> arr
a\b   b0   b1
a0  0.0  1.0
a1  2.0  3.0
>>> arr.info
2 x 2
a [2]: 'a0' 'a1'
b [2]: 'b0' 'b1'
dtype: float64

```

- To create a 1D array using `from_string()` and the default separator “`\t`”, a tabulation character `\t` (instead of – previously) must be added in front of the data line:

```

>>> from_string('sex M F
...             \t 0 1')
sex  M  F
    0  1

```

- viewer window title also includes the dtype of the current displayed array (closes [issue 85](#))
- viewer window title uses only the file name instead of the entire file path as it made titles too long in some cases.
- when editing .csv files, the viewer window title will be “directoryfname.csv - axes_info” instead of having the file name repeated as before (“dirfname.csv - fname: axes_info”).
- the viewer will not update digits/scientific notation nor colors when the filter changes, so that numbers are more easily comparable when quickly changing the filter, especially using the scrollwheel on filter boxes.
- NaN values display as grey in the viewer so that they stand out more.
- `compare()` will color values depending on relative difference instead of absolute difference as this is usually more useful.
- `compare(sessions)` uses `nan_equal` to compare arrays so that identical arrays are not marked different when they contain NaN values.
- changed `compare()` “stacked axis” names: arrays -> array and sessions -> session because that reads a bit more naturally.

Fixes

- fixed array creation with `axis(es)` given as string containing only one label (axis name and label were inverted).
- fixed reading an array from a CSV or Excel file when the columns axis is not explicitly named (via `\`). For example, let's say we want to read a CSV file 'pop.csv' with the following content (indented for clarity)

```
sex, 2015, 2016
  F,   11,   13
  M,   12,   10
```

The result of function `read_csv` is:

```
>>> pop = read_csv('pop.csv')
>>> pop
sex\{1}  2015  2016
      F    11    13
      M    12    10
```

Closes [issue 372](#).

- fixed converting a 1xN Pandas DataFrame to an array using `aslarray` (closes [issue 427](#)):

```
>>> df = pd.DataFrame([[1, 2, 3]], index=['a0'], columns=['b0', 'b1', 'b2'])
>>> df
   b0  b1  b2
a0   1   2   3
>>> aslarray(df)
{0}\{1}  b0  b1  b2
      a0   1   2   3
```

```
>>> # setting name to index and columns
>>> df.index.name = 'a'
>>> df.columns.name = 'b'
>>> df
   b  b0  b1  b2
a
a0   1   2   3
>>> aslarray(df)
a\b  b0  b1  b2
   a0   1   2   3
```

- fixed original file being deleted when trying to overwrite a file via `Session.save` or `open_excel` failed (closes [issue 441](#))
- fixed loading arrays from Excel sheets containing blank cells below or right of the array to read (closes [issue 443](#))
- fixed unary and binary operations between sessions failing entirely when the operation failed/was invalid on any array. Now the result will be nan for that array but the operation will carry on for other arrays.
- fixed stacking sessions failing entirely when the stacking failed on any array. Now the result will be nan for that array but the operation will carry on for other arrays.
- fixed stacking arrays with anonymous axes.
- fixed applying `split_axes` on an array with labels of type 'Object' (could happen when an array is read from a file).
- fixed background color in the viewer when using filters in the `compare()` dialog (closes [issue 66](#))

- fixed autoresize of columns by double clicking between column headers (closes [issue 43](#))
- fixed representing a 0D array (scalar) in the viewer (closes [issue 71](#))
- fixed viewer not displaying an error message when saving or loading a file failed (closes [issue 75](#))
- fixed `array.split_axis` when the combined axis does not contain all the combination of labels resulting from the split (closes [issue 369](#)).
- fixed `array.split_axis` when combined labels are not sorted by the first part then second part (closes [issue 364](#)).
- fixed opening .csv files in the editor will create variables named using only the filename without extension (instead of being named using the full path of the file – making it almost useless). Closes [issue 90](#).
- fixed deleting a variable (using the del key in the list) not marking the session/file as being modified.
- fixed the link to the tutorial (Help->Online Tutorial) (closes [issue 92](#)).
- fixed inplace modifications of arrays in the console (via `array[xxx] = value`) not updating the view (closes [issue 94](#)).
- fixed background color in `compare()` being wrong after changing axes order by drag-and-dropping them (closes [issue 89](#)).
- fixed the whole array/compare being the same color in the presence of `-inf` or `+inf` in the array.

6.1.9 Version 0.25.2

Released on 2017-09-06.

Miscellaneous improvements

- Excel Workbooks opened with `open_excel(visible=False)` will use the global Excel instance by default and those using `visible=True` will use a new Excel instance by default (closes [issue 405](#)).

Fixes

- fixed `view()` which did not show any array (closes [issue 57](#)).
- fixed exceptions in the viewer crashing it when a Qt app was created (e.g. from a plot) before the viewer was started (closes [issue 58](#)).
- fixed `compare()` arrays names not being determined correctly (closes [issue 61](#)).
- fixed filters and title not being updated when displaying array created via the console (closes [issue 55](#)).
- fixed array grid not being updated when selecting a variable when no variable was selected (closes [issue 56](#)).
- fixed copying or plotting multiple rows in the editor when they were selected via drag and drop on headers (closes [issue 59](#)).
- fixed digits not being automatically updated when changing filters.

6.1.10 Version 0.25.1

Released on 2017-09-04.

Miscellaneous improvements

- Deprecated methods display a warning message when they are still used (replaced DeprecationWarning by FutureWarning). Closes [issue 310](#).
- updated documentation of method `with_total` (closes [issue 89](#)).
- trying to set values of a subset by passing an array with incompatible axes displays a better error message (closes [issue 268](#)).

Fixes

- fixed error raised in viewer when switching between arrays when a filter was set.
- fixed displaying empty array when starting the viewer or a new session in it.
- fixed Excel instance created via `to_excel()` and `open_excel()` without any filename being closed at the end of the Python program (closes [issue 390](#)).
- fixed the `view()`, `edit()` and `compare()` functions not being available in the viewer console.
- fixed row and column resizing by double clicking on the edge of an header cell.
- fixed *New* and *Open* in the menu *File* of the viewer when IPython console is not available.
- fixed getting a subset of an array by mixing boolean filters and other filters (closes [issue 246](#)):

```
>>> arr = ndrange('a=a0..a2;b=0..3')
>>> arr
a\b  0  1  2  3
a0   0  1  2  3
a1   4  5  6  7
a2   8  9 10 11
>>> arr['a0,a2', x.b < 2]
a\b  0  1
a0   0  1
a2   8  9
```

Warning: when mixed with other filters, boolean filters are limited to one dimension.

- fixed setting an array values using `array.points[key] = value` when value is an LArray (closes [issue 368](#)).
- fixed using syntax `'int..int'` in a selection (closes [issue 350](#)):

```
>>> arr = ndrange('a=2017..2012')
>>> arr
a  2017  2016  2015  2014  2013  2012
   0     1     2     3     4     5
>>> arr['2012..2015']
a  2012  2013  2014  2015
   5     4     3     2
```

- fixed mixing `'..'` sequences and spaces in an indexing string (closes [issue 389](#)):

```
>>> arr = ndtest(7)
>>> arr
a  a0  a1  a2  a3  a4  a5  a6
   0   1   2   3   4   5   6
>>> arr['a0, a2, a4..a6']
a  a0  a2  a4  a5  a6
   0   2   4   5   6
```

- fixed indexing/aggregating using groups with renaming (using >>) when the axis has mixed type labels (object dtype).

6.1.11 Version 0.25

Released on 2017-08-22.

New features

- viewer functions (*view*, *edit* and *compare*) have been moved to the separate *larray-editor* package, which needs to be installed separately, unless you are using *larrayenv*. Closes [issue 332](#).
- installing *larray-editor* (or *larrayenv*) from conda environment creates a new menu ‘LArray’ in the Windows start menu. It contains a link to open the documentation, a shortcut to launch the user interface in edition mode and a shortcut to update *larrayenv*. Closes [issue 281](#).
- added possibility to transpose an array in the viewer by dragging and dropping axes’ names in the filter bar.
- implemented `array.align(other_array)` which makes two arrays compatible with each other (by making all common axes compatible). This is done by adding, removing or reordering labels for each common axis according to the join method used:
 - outer: will use a label if it is in either arrays axis (ordered like the first array). This is the default as it results in no information loss.
 - inner: will use a label if it is in both arrays axis (ordered like the first array)
 - left: will use the first array axis labels
 - right: will use the other array axis labels

The fill value for missing labels defaults to nan.

```
>>> arr1 = ndtest((2, 3))
>>> arr1
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
>>> arr2 = -ndtest((3, 2))
>>> # reorder array to make the test more interesting
>>> arr2 = arr2[['b1', 'b0']]
>>> arr2
a\\b  b1  b0
a0  -1   0
a1  -3  -2
a2  -5  -4
```

Align `arr1` and `arr2`


```

>>> aligned1, aligned2 = arr1.align(arr2)
>>> aligned1
a\b  b0  b1  b2
a0  0.0  1.0  2.0
a1  3.0  4.0  5.0
a2  nan  nan  nan
>>> aligned2
a\b  b0  b1  b2
a0  0.0 -1.0  nan
a1 -2.0 -3.0  nan
a2 -4.0 -5.0  nan

```

After aligning all common axes, one can then do operations between the two arrays

```

>>> aligned1 + aligned2
a\b  b0  b1  b2
a0  0.0  0.0  nan
a1  1.0  1.0  nan
a2  nan  nan  nan

```

The fill value for missing labels defaults to nan but can be changed to any compatible value.

```

>>> aligned1, aligned2 = arr1.align(arr2, fill_value=0)
>>> aligned1
a\b  b0  b1  b2
a0   0   1   2
a1   3   4   5
a2   0   0   0
>>> aligned2
a\b  b0  b1  b2
a0   0  -1   0
a1  -2  -3   0
a2  -4  -5   0
>>> aligned1 + aligned2
a\b  b0  b1  b2
a0   0   0   2
a1   1   1   5
a2  -4  -5   0

```

- implemented `Session.transpose(axes)` to reorder axes of all arrays within a session, ignoring missing axes for each array. For example, let us first create a test session and a small helper function to display sessions as a short summary.

```

>>> arr1 = ndtest((2, 2, 2))
>>> arr2 = ndtest((2, 2))
>>> sess = Session([('arr1', arr1), ('arr2', arr2)])
>>> def print_summary(s):
...     print(s.summary("{name} -> {axes_names}"))
>>> print_summary(sess)
arr1 -> a, b, c
arr2 -> a, b

```

Put the 'b' axis in front of all arrays

```

>>> print_summary(sess.transpose('b'))
arr1 -> b, a, c
arr2 -> b, a

```

Axes missing on an array are ignored ('c' for arr2 in this case)

```
>>> print_summary(sess.transpose('c', 'b'))
arr1 -> c, b, a
arr2 -> b, a
```

Use ... to move axes to the end

```
>>> print_summary(sess.transpose(..., 'a'))
arr1 -> b, c, a
arr2 -> b, a
```

- implemented unary operations on Session, which means one can negate all arrays in a Session or take the absolute value of all arrays in a Session without writing an explicit loop for that.

```
>>> arr1 = ndtest(2)
>>> arr1
a  a0  a1
   0   1
>>> arr2 = ndtest(4) - 1
>>> arr2
a  a0  a1  a2  a3
   -1   0   1   2
>>> sess1 = Session([('arr1', arr1), ('arr2', arr2)])
>>> sess2 = -sess1
>>> sess2.arr1
a  a0  a1
   0  -1
>>> sess2.arr2
a  a0  a1  a2  a3
   1   0  -1  -2
>>> sess3 = abs(sess1)
>>> sess3.arr2
a  a0  a1  a2  a3
   1   0   1   2
```

- implemented stacking sessions using stack().

Let us first create two test sessions. For example suppose we have a session storing the results of a baseline simulation:

```
>>> arr1 = ndtest(2)
>>> arr1
a  a0  a1
   0   1
>>> arr2 = ndtest(3)
>>> arr2
a  a0  a1  a2
   0   1   2
>>> baseline = Session([('arr1', arr1), ('arr2', arr2)])
```

and another session with a variant

```
>>> arr1variant = arr1 * 2
>>> arr1variant
a  a0  a1
   0   2
>>> arr2variant = 2 - arr2 / 2
```

(continues on next page)

(continued from previous page)

```
>>> arr2variant
a  a0  a1  a2
   2.0 1.5 1.0
>>> variant = Session([('arr1', arr1variant), ('arr2', arr2variant)])
```

then we stack them together

```
>>> stacked = stack([('baseline', baseline), ('variant', variant)], 'sessions')
>>> stacked
Session(arr1, arr2)
>>> stacked.arr1
a\sessions baseline variant
      a0         0         0
      a1         1         2
>>> stacked.arr2
a\sessions baseline variant
      a0         0.0        2.0
      a1         1.0        1.5
      a2         2.0        1.0
```

Combined with the fact that we can compute some very simple expressions on sessions, this can be extremely useful to quickly compare all arrays of several sessions (e.g. simulation variants):

```
>>> diff = variant - baseline
>>> # compute the absolute difference and relative difference for each array of
    ↪ the sessions
>>> stacked = stack([('baseline', baseline),
                    ('variant', variant),
                    ('diff', diff),
                    ('abs diff', abs(diff)),
                    ('rel diff', diff / baseline)], 'sessions')
>>> stacked
Session(arr1, arr2)
>>> stacked.arr2
a\sessions baseline variant diff abs diff rel diff
      a0         0.0        2.0   2.0     2.0     inf
      a1         1.0        1.5   0.5     0.5     0.5
      a2         2.0        1.0  -1.0     1.0    -0.5
```

- implemented `Axis.align(other_axis)` and `AxisCollection.align(other_collection)` which makes two axes / axis collections compatible with each other, see `LArray.align` above.
- implemented `Session.apply(function)` to apply a function to all elements (arrays) of a `Session` and return a new `Session`.

Let us first create a test session

```
>>> arr1 = ndtest(2)
>>> arr1
a  a0  a1
   0   1
>>> arr2 = ndtest(3)
>>> arr2
a  a0  a1  a2
   0   1   2
>>> sess1 = Session([('arr1', arr1), ('arr2', arr2)])
```

(continues on next page)

(continued from previous page)

```
>>> sess1
Session(arr1, arr2)
```

Then define the function we want to apply to all arrays of our session

```
>>> def increment(element):
...     return element + 1
```

Apply it

```
>>> sess2 = sess1.apply(increment)
>>> sess2.arr1
a  a0  a1
   1   2
>>> sess2.arr2
a  a0  a1  a2
   1   2   3
```

- implemented setting the value of multiple points using `array.points[labels] = value`

```
>>> arr = ndtest((3, 4))
>>> arr
a\b  b0  b1  b2  b3
a0   0   1   2   3
a1   4   5   6   7
a2   8   9  10  11
```

Now, suppose you want to retrieve several specific combinations of labels, for example (a0, b1), (a0, b3), (a1, b0) and (a2, b2). You could write a loop like this:

```
>>> values = []
>>> for a, b in [('a0', 'b1'), ('a0', 'b3'), ('a1', 'b0'), ('a2', 'b2')]:
...     values.append(arr[a, b])
>>> values
[1, 3, 4, 10]
```

but you could also (this already worked in previous versions) use `array.points` like:

```
>>> arr.points[['a0', 'a0', 'a1', 'a2'], ['b1', 'b3', 'b0', 'b2']]
a,b  a0,b1  a0,b3  a1,b0  a2,b2
      1       3       4      10
```

which has the advantages of being both much faster and keep more information. Now suppose you want to *set* the value of those points, you could write:

```
>>> for a, b in [('a0', 'b1'), ('a0', 'b3'), ('a1', 'b0'), ('a2', 'b2')]:
...     arr[a, b] = 42
>>> arr
a\b  b0  b1  b2  b3
a0   0  42   2  42
a1  42   5   6   7
a2   8   9  42  11
```

but now you can also use the faster alternative:

```
>>> arr.points[['a0', 'a0', 'a1', 'a2'], ['b1', 'b3', 'b0', 'b2']] = 42
```

Miscellaneous improvements

- added icon to display in Windows start menu and editor windows.
- viewer keeps labels visible even when scrolling (label rows and columns are now frozen).
- added ‘Getting Started’ section in documentation.
- implemented axes argument to ipfp to specify on which axes the fitting procedure should be applied (closes [issue 185](#)). For example, let us assume you have a 3D array, such as:

```
>>> initial = ndrange('a=a0..a9;b=b0..b9;year=2000..2016')
```

and you want to apply a 2D fitting procedure for each value of the year axis. Previously, you had to loop on that year axis explicitly and call ipfp within the loop, like:

```
>>> result = zeros(initial.axes)
>>> for year in initial.year:
...     current = initial[year]
...     # assume you have some targets for each year
...     current_targets = [current.sum(x.a) + 1, current.sum(x.b) + 1]
...     result[year] = ipfp(current_targets, current)
```

Now you can apply the procedure on all years at once, by telling you want to do the fitting procedure on the other axes. This is a bit shorter to type, but this is also *much* faster.

```
>>> all_targets = [initial.sum(x.a) + 1, initial.sum(x.b) + 1]
>>> result = ipfp(all_targets, initial, axes=(x.a, x.b))
```

- made ipfp 10 to 20% faster (even without using the axes argument).
- implemented Session.to_globals(inplace=True) which will update the content of existing arrays instead of creating new variables and overwriting them. This ensures the arrays have the same axes in the session than the existing variables.
- added the ability to provide a pattern when loading several .csv files as a session. Among others, patterns can use * to match any number of characters and ? to match any single character.

```
>>> s = Session()
>>> # load all .csv files starting with "output" in the data directory
>>> s.load('data/output*.csv')
```

- stack can be used with keyword arguments when labels are “simple strings” (i.e. no integers, no punctuation, no string starting with integers, etc.). This is an attractive alternative but as it only works in the usual case and not in all cases, it is not recommended to use it except in the interactive console.

```
>>> arr1 = ones('nat=BE,FO')
>>> arr1
nat  BE  FO
    1.0  1.0
>>> arr2 = zeros('nat=BE,FO')
>>> arr2
nat  BE  FO
    0.0  0.0
>>> stack(M=arr1, F=arr2, axis='sex=M,F')
nat\\sex  M  F
    BE  1.0  0.0
    FO  1.0  0.0
```

Without passing an explicit order for labels like above (or an axis object), it should only be used on Python 3.6 or later because keyword arguments are NOT ordered on earlier Python versions.

```
>>> # use this only on Python 3.6 and later
>>> stack(M=arr1, F=arr2, axis='sex')
nat\\sex      M      F
    BE  1.0  0.0
    FO  1.0  0.0
```

- binary operations between session now ignore type errors. For example, if you are comparing two sessions with many arrays by computing the difference between them but a few arrays contain strings, the whole operation will not fail, the concerned arrays will be assigned a nan instead.
- added optional argument *ignore_exceptions* to Session.load to ignore exceptions during load. This is mostly useful when trying to load many .csv files in a Session and some of them have an invalid format but you want to load the others.

Fixes

- fixed disambiguating an ambiguous key by adding the axis within the string, for example arr['axis_name[ambiguouslabel]'] (closes [issue 331](#)).
- fixed converting a string group to integer or float using int() and float() (when that makes sense).

```
>>> a = Axis('a=10,20,30,total')
>>> a
Axis(['10', '20', '30', 'total'], 'a')
>>> str(a.i[0])
'10'
>>> int(a.i[0])
10
>>> float(a.i[0])
10.0
```

6.1.12 Version 0.24.1

Released on 2017-06-14.

Fixes

- updated the tutorial to use version 0.24 syntax.

6.1.13 Version 0.24

Released on 2017-06-14.

New features

- implemented `Session.to_globals` which creates global variables from variables stored in the session (closes [issue 276](#)). Note that this should usually only be used in an interactive console and not in a script. Code editors are confused by this kind of manipulation and will likely consider as invalid the code using variables created in this way. Additionally, when using this method auto-completion, “show definition”, “go to declaration” and other similar code editor features will probably not work for the variables created in this way and any variable derived from them.

```
>>> s = Session(arr1=ndtest(3), arr2=ndtest((2, 2)))
>>> s.to_globals()
>>> arr1
a  a0  a1  a2
   0   1   2
>>> arr2
a\b  b0  b1
   a0   0   1
   a1   2   3
```

- added new boolean argument ‘`overwrite`’ to `Session.save`, `Session.to_hdf`, `Session.to_excel` and `Session.to_pickle` methods (closes [issue 293](#)). If `overwrite=True` and the target file already existed, it is deleted and replaced by a new one. This is the new default behavior. If `overwrite=False`, an existing file is updated (like it was in previous larray versions):

```
>>> arr1, arr2, arr3 = ndtest((2, 2)), ndtest(4), ndtest((3, 2))
>>> s = Session([('arr1', arr1), ('arr2', arr2), ('arr3', arr3)])
```

```
>>> # save arr1, arr2 and arr3 in file output.h5
>>> s.save('output.h5')
```

```
>>> # replace arr1 and create arr4 + put them in an second session
>>> arr1, arr4 = ndtest((3, 3)), ndtest((2, 3))
>>> s2 = Session([('arr1', arr1), ('arr4', arr4)])
```

```
>>> # replace arr1 and add arr4 in file output.h5
>>> s2.save('output.h5', overwrite=False)
```

```
>>> # erase content of 'output.h5' and save only arrays contained in the second_
↪session
>>> s2.save('output.h5')
```

Miscellaneous improvements

- renamed `create_sequential()` to `sequence()` (closes [issue 212](#)).
- improved auto-completion in ipython interactive consoles (e.g. the viewer console) for `Axis`, `AxisCollection`, `Group` and `Workbook` objects. These objects can now complete keys within `[]`.

```
>>> gender = Axis('gender=Male,Female')
>>> gender
Axis(['Male', 'Female'], 'gender')
gender['Female
>>> gender['Fe<tab> # will be completed to `gender['Female`
```

```
>>> arr = ndrange(gender)
>>> arr.axes['gender'] # will be completed to `arr.axes['gender`
```

```
>>> wb = open_excel()
>>> wb['Sheet1'] # will be completed to `wb['Sheet1`
```

- added documentation for Session methods (closes [issue 277](#)).
- allowed to provide explicit names for arrays or sessions in compare(). Closes [issue 307](#).

Fixes

- fixed title argument of *ndtest* creation function: title was not passed to the returned array.
- fixed create_sequential when arguments initial and inc are array and scalar respectively (closes [issue 288](#)).
- fixed auto-completion of attributes of LArray and Group objects (closes [issue 302](#)).
- fixed name of arrays/sessions in compare() not being inferred correctly (closes [issue 306](#)).
- fixed indexing Excel sheets by position to always yield the requested shape even when bounds are outside the range of used cells. Closes [issue 273](#).
- fixed the array() method on excel.Sheet returning float labels when int labels are expected.
- fixed getting float data instead of int when converting an Excel Sheet or Range to an larray or numpy array.
- fixed some warning messages to point to the correct line in user code.
- fixed crash of Session.save method when it contained 0D arrays. They are now skipped when saving a session (closes [issue 291](#)).
- fixed Session.save and Session.to_excel failing to create new Excel files (it only worked if the file already existed). Closes [issue 313](#).
- fixed Session.load(file, engine='pandas_excel') : axes were considered as anonymous.

6.1.14 Version 0.23

Released on 2017-05-30.

Miscellaneous improvements

- changed display of arrays (closes [issue 243](#)):

```
>>> ndtest((2, 3))
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
```

instead of

```
>>> ndtest((2, 3))
a\b | b0 | b1 | b2
a0 |  0 |  1 |  2
a1 |  3 |  4 |  5
```


- .. can now be used within keys (between []). Previously it could only be used to define new axes. As a reminder, it generates increasing values between the two bounds. It is slightly different from : which takes everything between the two bounds **in the axis order**.

```
>>> arr = ndrange('a=a1,a0,a2,a3')
>>> arr
a  a1  a0  a2  a3
   0   1   2   3
>>> arr['a1..a3']
a  a1  a2  a3
   0   2   3
```

this is different from : which takes everything in between the two bounds :

```
>>> arr['a1:a3']
a  a1  a0  a2  a3
   0   1   2   3
```

- in both axes definitions and keys (within []) .. can now be mixed with , and other .. :

```
>>> arr = ndrange('code=A,C..E,G,X..Z')
>>> arr
code  A  C  D  E  G  X  Y  Z
      0  1  2  3  4  5  6  7
>>> arr['A,Z..X,G']
code  A  Z  Y  X  G
      0  7  6  5  4
```

- within .. extra zeros are only padded to numbers if zeros are present in the pattern.

```
>>> ndrange('code=A1..A12')
code  A1  A2  A3  A4  A5  A6  A7  A8  A9  A10  A11  A12
      0   1   2   3   4   5   6   7   8   9   10   11
```

```
>>> ndrange('code=A01..A12')
code  A01  A02  A03  A04  A05  A06  A07  A08  A09  A10  A11  A12
      0    1    2    3    4    5    6    7    8    9   10   11
```

in previous larray versions, the two above definitions returned the second array.

- set *sep* argument of from_string function to ‘ ‘ by default (closes [issue 271](#)). For 1D array, a “-” must be added in front of the data line.

```
>>> from_string('''sex  M  F
                  -   0  1''')
sex  M  F
    0  1
>>> from_string('''nat\\sex  M  F
                  BE    0  1
                  FO    2  3''')
nat\\sex  M  F
        BE  0  1
        FO  2  3
```

- improved error message when trying to access nonexistent sheet in an Excel workbook (closes [issue 266](#)).
- when creating an Axis from a Group and no explicit name was given, reuse the name of the group axis.

```
>>> a = Axis('a=a0..a2')
>>> Axis(a[:'a1'])
Axis(['a0', 'a1'], 'a')
```

- allowed to create an array using a single group as if it was an Axis.

```
>>> a = Axis('a=a0..a2')
>>> ndrange(a)
a  a0  a1  a2
   0   1   2
>>> # using a group as an axis
>>> ndrange(a[:'a1'])
a  a0  a1
   0   1
```

- allowed to use axes (Axis objects) to subset arrays (part of [issue 210](#)).

```
>>> arr = ndtest((2, 3))
>>> arr
a\b  b0  b1  b2
a0    0   1   2
a1    3   4   5
>>> b2 = Axis('b=b0,b2')
>>> arr[b2]
a\b  b0  b2
a0    0   2
a1    3   5
```

- improved string representation of Excel workbooks and sheets (they mention the actual file/sheet they correspond to). This is mostly useful in the interactive console to check what an object corresponds to.

```
>>> wb = open_excel()
>>> wb
<larray.io.excel.Workbook [Book1]>
>>> wb[0]
<larray.io.excel.Sheet [Book1]Sheet1>
```

Fixes

- `open_excel('non existent file')` will raise an explicit error immediately when `overwrite_file` is `False`, instead of failing at a seemingly random point later on (closes [issue 265](#)).
- integer-like strings in axis definition strings using `,` are converted to integers to be consistent with string definitions using `...`. In other words, `ndrange('a=1,2,3')` did not create the same array than `ndrange('a=1..3')`.
- fixed reading a single cell from an Excel sheet.
- fixed script execution not resuming after quitting the viewer when it was called using `view(a_single_array)`.
- fixed opening the viewer after showing a plot window.
- do not display an error when setting the value of an element of a non LArray sequence in the viewer console

```
>>> l = [1, 2, 3]
>>> l[0] = 42
```

6.1.15 Version 0.22

Released on 2017-05-11.

New features

- viewer: added a menu bar with the ability to clear the current session, save all its arrays to a file (.h5, .xlsx, or a directory containing multiple .csv files), and load arrays from such a file (closes [issue 88](#)).

WARNING: Only array objects are currently saved. It means that scalars, functions or others non-LArray objects defined in the console are *not* saved in the file.

- implemented a new describe() method on arrays to give quick summary statistics. By default, it includes the number of non-NaN values, the mean, standard deviation, minimum, 25, 50 and 75 percentiles and maximum.

```
>>> arr = ndrange('gender=Male,Female;year=2014..2020').astype(float)
>>> arr
gender\year | 2014 | 2015 | 2016 | 2017 | 2018 | 2019 | 2020
      Male |  0.0 |  1.0 |  2.0 |  3.0 |  4.0 |  5.0 |  6.0
      Female |  7.0 |  8.0 |  9.0 | 10.0 | 11.0 | 12.0 | 13.0
>>> arr.describe()
statistic | count | mean |          std | min | 25% | 50% | 75% | max
          | 14.0 |  6.5 | 4.031128874149275 | 0.0 | 3.25 | 6.5 | 9.75 | 13.0
```

an optional keyword argument allows to specify different percentiles to include

```
>>> arr.describe(percentiles=[20, 40, 60, 80])
statistic | count | mean |          std | min | 20% | 40% | 60% | 80% | max
          | 14.0 |  6.5 | 4.031128874149275 | 0.0 | 2.6 | 5.2 | 7.8 | 10.4 | 13.0
```

its sister method, describe_by() was also implemented to give quick summary statistics along axes or groups.

```
>>> arr.describe_by('gender')
gender\statistic | count | mean | std | min | 25% | 50% | 75% | max
      Male |  7.0 |  3.0 | 2.0 | 0.0 | 1.5 | 3.0 | 4.5 | 6.0
      Female |  7.0 | 10.0 | 2.0 | 7.0 | 8.5 | 10.0 | 11.5 | 13.0
>>> arr.describe_by('gender', (x.year[:2015], x.year[2019:]))
gender | year\statistic | count | mean | std | min | 25% | 50% | 75% | max
      Male |      :2015 |  2.0 |  0.5 | 0.5 | 0.0 | 0.25 | 0.5 | 0.75 | 1.0
      Male |      2019: |  2.0 |  5.5 | 0.5 | 5.0 | 5.25 | 5.5 | 5.75 | 6.0
      Female |      :2015 |  2.0 |  7.5 | 0.5 | 7.0 | 7.25 | 7.5 | 7.75 | 8.0
      Female |      2019: |  2.0 | 12.5 | 0.5 | 12.0 | 12.25 | 12.5 | 12.75 | 13.0
```

This closes [issue 184](#).

- implemented reindex allowing to change the order of labels and add/remove some of them to one or several axes:

```
>>> arr = ndtest((2, 2))
>>> arr
a\b | b0 | b1
a0 |  0 |  1
a1 |  2 |  3
>>> arr.reindex(x.b, ['b1', 'b2', 'b0'], fill_value=-1)
a\b | b1 | b2 | b0
a0 |  1 | -1 |  0
a1 |  3 | -1 |  2
```

(continues on next page)

(continued from previous page)

```
>>> a = Axis('a', ['a1', 'a2', 'a0'])
>>> b = Axis('b', ['b2', 'b1', 'b0'])
>>> arr.reindex({'a': a, 'b': b}, fill_value=-1)
a\b | b2 | b1 | b0
a1 | -1 | 3 | 2
a2 | -1 | -1 | -1
a0 | -1 | 1 | 0
```

using `reindex` one can make an array compatible with another array which has more/less labels or with labels in a different order:

```
>>> arr2 = ndtest((3, 3))
>>> arr2
a\b | b0 | b1 | b2
a0 | 0 | 1 | 2
a1 | 3 | 4 | 5
a2 | 6 | 7 | 8
>>> arr.reindex(arr2.axes, fill_value=0)
a\b | b0 | b1 | b2
a0 | 0 | 1 | 0
a1 | 2 | 3 | 0
a2 | 0 | 0 | 0
>>> arr.reindex(arr2.axes, fill_value=0) + arr2
a\b | b0 | b1 | b2
a0 | 0 | 2 | 2
a1 | 5 | 7 | 5
a2 | 6 | 7 | 8
```

This closes [issue 18](#).

- added `load_example_data` function to load datasets used in tutorial and be able to reproduce examples. The name of the dataset must be provided as argument (there is currently only one available dataset). Datasets are returned as Session objects:

```
>>> demo = load_example_data('demography')
>>> demo.pop.info
26 x 3 x 121 x 2 x 2
time [26]: 1991 1992 1993 ... 2014 2015 2016
geo [3]: 'BruCap' 'Fla' 'Wal'
age [121]: 0 1 2 ... 118 119 120
sex [2]: 'M' 'F'
nat [2]: 'BE' 'FO'
>>> demo.qx.info
26 x 3 x 121 x 2 x 2
time [26]: 1991 1992 1993 ... 2014 2015 2016
geo [3]: 'BruCap' 'Fla' 'Wal'
age [121]: 0 1 2 ... 118 119 120
sex [2]: 'M' 'F'
nat [2]: 'BE' 'FO'
```

(closes [issue 170](#))

- implemented `Axis.union`, `intersection` and `difference` which produce new axes by combining the labels of the axis with the other labels.

```
>>> letters = Axis('letters=a,b')
>>> letters.union(Axis('letters=b,c'))
```

(continues on next page)

(continued from previous page)

```
Axis(['a', 'b', 'c'], 'letters')
>>> letters.union(['b', 'c'])
Axis(['a', 'b', 'c'], 'letters')
>>> letters.intersection(['b', 'c'])
Axis(['b'], 'letters')
>>> letters.difference(['b', 'c'])
Axis(['a'], 'letters')
```

- implemented `Group.union`, `intersection` and `difference` which produce new groups by combining the labels of the group with the other labels.

```
>>> letters = Axis('letters=a..d')
>>> letters['a', 'b'].union(letters['b', 'c'])
letters['a', 'b', 'c'].set()
>>> letters['a', 'b'].union(['b', 'c'])
letters['a', 'b', 'c'].set()
>>> letters['a', 'b'].intersection(['b', 'c'])
letters['b'].set()
>>> letters['a', 'b'].difference(['b', 'c'])
letters['a'].set()
```

- viewer: added possibility to delete an array by pressing Delete on keyboard (closes [issue 116](#)).
- Excel sheets in workbooks opened via `open_excel` can be renamed by changing their `.name` attribute:

```
>>> wb = open_excel()
>>> wb['old_sheet_name'].name = 'new_sheet_name'
```

- Excel sheets in workbooks opened via `open_excel` can be deleted using “del”:

```
>>> wb = open_excel()
>>> del wb['sheet_name']
```

- implemented `PGroup.set()` to transform a positional group to an `LSet`.

```
>>> a = Axis('a=a0..a5')
>>> a.i[:2].set()
a['a0', 'a1'].set()
```

Miscellaneous improvements

- inverted `name` and `labels` arguments when creating an `Axis` and made `name` argument optional (to create anonymous axes). Now, it is also possible to create an `Axis` by passing a single string of the kind ‘name=labels’:

```
>>> anonymous = Axis('0..100')
>>> age = Axis('age=0..100')
>>> gender = Axis('M,F', 'gender')
```

(closes [issue 152](#))

- renamed `Session.dump`, `dump_hdf`, `dump_excel` and `dump_csv` to `save`, `to_hdf`, `to_excel` and `to_csv` (closes [issue 217](#)).
- changed default value of `ddof` argument for `var` and `std` functions from 0 to 1 (closes [issue 190](#)).
- implemented a new syntax for `stack()`: `stack({label1: value1, label2: value2}, axis)`

```
>>> nat = Axis('nat', 'BE, FO')
>>> sex = Axis('sex', 'M, F')
>>> males = ones(nat)
>>> males
nat | BE | FO
   | 1.0 | 1.0
>>> females = zeros(nat)
>>> females
nat | BE | FO
   | 0.0 | 0.0
```

In the case the axis has already been defined in a variable, this gives:

```
>>> stack({'M': males, 'F': females}, sex)
nat\sex | M | F
      BE | 1.0 | 0.0
      FO | 1.0 | 0.0
```

Additionally, axis can now be an axis string definition in addition to an Axis object, which means one can write this:

```
>>> stack({'M': males, 'F': females}, 'sex=M,F')
```

It is better than the simpler but *highly discouraged* alternative:

```
>>> stack([males, females], sex)
```

because it is all too easy to invert labels. It is very hard to spot the error in the following line, and larray cannot spot it for you either:

```
>>> stack([females, males], sex)
nat\sex | M | F
      BE | 0.0 | 1.0
      FO | 0.0 | 1.0
```

When creating an axis from scratch (it does not already exist in a variable), one might want to use this:

```
>>> stack([males, females], 'sex=M,F')
```

even if this could suffer, to a lesser extent, the same problem as above when stacking many arrays.

- handle ... in transpose method to avoid having to list all axes. This can be useful, for example, to change which axis is displayed in columns (closes [issue 188](#)).

```
>>> arr.transpose(..., 'time')
>>> arr.transpose('gender', ..., 'time')
```

- made scalar Groups behave even more like their value: any method available on the value is available on the Group. For example, if the Group has a string value, the string methods are available on it (closes [issue 202](#)).

```
>>> test = Axis('test', ['abc', 'a1-a2'])
>>> test.i[0].upper()
'ABC'
>>> test.i[1].split('-')
['a1', 'a2']
```

- updated AxisCollection.replace so as to replace one, several or all axes and to accept axis definitions as new axes.

```

>>> arr = ndtest((2, 3))
>>> axes = arr.axes
>>> axes
AxisCollection([
  Axis(['a0', 'a1'], 'a'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
>>> row = Axis(['r0', 'r1'], 'row')
>>> column = Axis(['c0', 'c1', 'c2'], 'column')

```

Replace several axes (keywords, list of tuple or dictionary)

```

>>> axes.replace(a=row, b=column)
>>> # or
>>> axes.replace(a="row=r0,r1", b="column=c0,c1,c2")
>>> # or
>>> axes.replace([(x.a, row), (x.b, column)])
>>> # or
>>> axes.replace({x.a: row, x.b: column})
AxisCollection([
  Axis(['r0', 'r1'], 'row'),
  Axis(['c0', 'c1', 'c2'], 'column')
])

```

- added possibility to delete an array from a session:

```

>>> s = Session({'a': ndtest((3, 3)), 'b': ndtest((2, 4)), 'c': ndtest((4, 2))})
>>> s.names
['a', 'b', 'c']
>>> del s.b
>>> del s['c']
>>> s.names
['a']

```

- made `create_sequential` axis argument accept axis definitions in addition to Axis objects like, for example, using a string definition (closes [issue 160](#)).

```

>>> create_sequential('year=2016..2019')
year | 2016 | 2017 | 2018 | 2019
    |    0 |    1 |    2 |    3

```

- replaced `*args`, `**kwargs` by explicit arguments in documentation of aggregation functions (sum, prod, mean, std, var, ...). Closes [issue 41](#).
- improved documentation of plot method (closes [issue 169](#)).
- improved auto-completion in ipython interactive consoles for both LArray and Session objects. LArray objects can now complete keys within `[]`.

```

>>> a = ndrange('sex=Male,Female')
>>> a
sex | Male | Female
    |    0 |    1
>>> a['Fe<tab>`

```

will autocomplete to `a['Female`. Sessions will now auto-complete both attributes (using `session.`) and keys (using `session[]`).

```
>>> s = Session({'a_nice_test_array': ndtest(10)})
>>> s.a_<tab>
```

will autocomplete to `s.a_nice_test_array` and `sf'a_<tab>` will be completed to `sf'a_nice_test_array`

- made warning messages for division by 0 and invalid values (usually caused by 0 / 0) point to the user code line, instead of the corresponding line in the larray module.
- preserve order of arrays in a session when saving to/loading from an .xlsx file.
- when creating a session from a directory containing CSV files, the directory may now contain other (non-CSV) files.
- several calls to `open_excel` from within the same program/script will now reuses a single global Excel instance. This makes Excel I/O much faster without having to create an instance manually using `xlwings.App`, and still without risking interfering with other instances of Excel opened manually (closes [issue 245](#)).
- improved error message when trying to copy a sheet from one instance of Excel to another (closes [issue 231](#)).

Fixes

- fixed keyword arguments such as `out`, `ddof`, ... for aggregation functions (closes [issue 189](#)).
- fixed `percentile(_by)` with multiple percentiles values, i.e. when argument `q` is a list/tuple (closes [issue 192](#)).
- fixed group aggregates on integer arrays for median, percentile, var and std (closes [issue 193](#)).
- fixed group sum over boolean arrays (closes [issue 194](#)).
- fixed `set_labels` when `inplace=True`.
- fixed array creation functions not raising an exception when called with wrong syntax `func(axis1, axis2, ...)` instead of `func([axis1, axis2, ...])` (closes [issue 203](#)).
- fixed position of added sheets in excel workbook: new sheets are appended instead of prepended (closes [issue 229](#)).
- fixed Workbook behavior in case of new workbook: the first added sheet replaces the default sheet *Sheet1* (closes [issue 230](#)).
- fixed name of Workbook sheets created by copying another sheet (closes [issue 244](#)).

```
>>> wb = open_excel()
>>> wb['name_of_new_sheet'] = wb['name_of_sheet_to_copy']
```

- fixed `with_axes` warning to refer to `set_axes` instead of `replace_axes`.
- fixed displayed title in viewer: shows path to file associated with current session + current array info + extra info (closes [issue 181](#))

6.1.16 Version 0.21

Released on 2017-03-28.

New features

- implemented `set_axes()` method to replace one, several or all axes of an array (closes [issue 67](#)). The method `with_axes()` is now deprecated (`set_axes()` must be used instead).

```
>>> arr = ndtest((2, 3))
>>> arr
a\b | b0 | b1 | b2
a0 |  0 |  1 |  2
a1 |  3 |  4 |  5
>>> row = Axis('row', ['r0', 'r1'])
>>> column = Axis('column', ['c0', 'c1', 'c2'])
```

Replace one axis (second argument *new_axis* must be provided)

```
>>> arr.set_axes(x.a, row)
row\b | b0 | b1 | b2
r0 |  0 |  1 |  2
r1 |  3 |  4 |  5
```

Replace several axes (keywords, list of tuple or dictionary)

```
>>> arr.set_axes(a=row, b=column)
or
>>> arr.set_axes([(x.a, row), (x.b, column)])
or
>>> arr.set_axes({x.a: row, x.b: column})
row\column | c0 | c1 | c2
r0 |  0 |  1 |  2
r1 |  3 |  4 |  5
```

Replace all axes (list of axes or `AxisCollection`)

```
>>> arr.set_axes([row, column])
row\column | c0 | c1 | c2
r0 |  0 |  1 |  2
r1 |  3 |  4 |  5
>>> arr2 = ndrange([row, column])
>>> arr.set_axes(arr2.axes)
row\column | c0 | c1 | c2
r0 |  0 |  1 |  2
r1 |  3 |  4 |  5
```

- implemented `Axis.replace` to replace some labels from an axis:

```
>>> sex = Axis('sex', ['M', 'F'])
>>> sex
Axis('sex', ['M', 'F'])
>>> sex.replace('M', 'Male')
Axis('sex', ['Male', 'F'])
>>> sex.replace({'M': 'Male', 'F': 'Female'})
Axis('sex', ['Male', 'Female'])
```

- implemented `from_string()` method to create an array from a string (closes [issue 96](#)).

```
>>> from_string(''age,nat\\sex, M, F
...           0,  BE,      0,  1
...           0,  FO,      2,  3
...           1,  BE,      4,  5
...           1,  FO,      6,  7'')
```

age	nat\\sex	M	F
0	BE	0	1
0	FO	2	3
1	BE	4	5
1	FO	6	7

- allowed to use a regular expression in `split_axis` method (closes [issue 106](#)):

```
>>> combined = ndrange('a_b = a0b0..a1b2')
>>> combined
a_b | a0b0 | a0b1 | a0b2 | a1b0 | a1b1 | a1b2
   |  0  |  1  |  2  |  3  |  4  |  5
>>> combined.split_axis(x.a_b, regex='(\\w{2})(\\w{2})')
a\\b | b0 | b1 | b2
a0  |  0 |  1 |  2
a1  |  3 |  4 |  5
```

- one can assign a new axis to several groups at the same time by using `axis[groups]`:

```
>>> group1 = year[2001:2004]
>>> group2 = year[2008,2009]
>>> # let us change the year axis by time
>>> x.time[group1, group2]
(x.time[2001:2004], x.time[2008, 2009])
```

- implemented `Axis.by()` which is equivalent to `axis[:].by()` and divides the axis into several groups of specified length:

```
>>> year = Axis('year', '2010..2016')
>>> year.by(3)
(year.i[0:3], year.i[3:6], year.i[6:7])
```

which is equivalent to `(year[2010:2012], year[2013:2015], year[2016])`. Like for groups, the optional second argument specifies the step between groups

```
>>> year.by(3, step=4)
(year.i[0:3], year.i[4:7])
```

which is equivalent to `(year[2010:2012], year[2014:2016])`. And if step is smaller than length, we get overlapping groups, which can be useful for example for moving averages.

```
>>> year.by(3, 2)
(year.i[0:3], year.i[2:5], year.i[4:7], year.i[6:7])
```

which is equivalent to `(year[2010:2012], year[2012:2014], year[2014:2016], year[2016])`

- implemented `larray_nan_equal` to test whether two arrays are identical even in the presence of nan values. Two arrays are considered identical by `larray_equal` if they have exactly the same axes and data. However, since a nan value has the odd property of not being equal to itself, `larray_equal` returns False if either array contains a nan value. `larray_nan_equal` returns True if all not-nan data is equal and both arrays have nans at the same place.

```

>>> arr1 = ndtest((2, 3), dtype=float)
>>> arr1['a1', 'b1'] = nan
>>> arr1
a\b | b0 | b1 | b2
a0 | 0.0 | 1.0 | 2.0
a1 | 3.0 | nan | 5.0
>>> arr2 = arr1.copy()
>>> arr2
a\b | b0 | b1 | b2
a0 | 0.0 | 1.0 | 2.0
a1 | 3.0 | nan | 5.0
>>> larray_equal(arr1, arr2)
False
>>> larray_nan_equal(arr1, arr2)
True
>>> arr2['b1'] = 0.0
>>> larray_nan_equal(arr1, arr2)
False

```

Miscellaneous improvements

- viewer: make keyboard shortcuts work even when the focus is not on the array editor widget. It means that, for example, plotting an array (via Ctrl-P) or opening it in Excel (Ctrl-E) can be done directly even when interacting with the list of arrays or within the interactive console (closes [issue 102](#)).
- viewer: automatically display plots done in the viewer console in a separate window (see example below), unless “%matplotlib inline” is used.

```

>>> arr = ndtest((3, 3))
>>> arr.plot()

```

- viewer: when calling `view(an_array)` from within the viewer, the new window opened does not block the initial window, which means you can have several windows open at the same time. `view()` without argument can still result in odd behavior though.
- improved `LArray.set_labels` to make it possible to replace only some labels of an axis, instead of all of them and to replace labels from several axes at the same time.

```

>>> a = ndrange('nat=BE,FO;sex=M,F')
>>> a
nat\sex | M | F
      BE | 0 | 1
      FO | 2 | 3

```

to replace only some labels, one must give a mapping giving the new label for each label to replace

```

>>> a.set_labels(x.sex, {'M': 'Men'})
nat\sex | Men | F
      BE | 0 | 1
      FO | 2 | 3

```

to replace labels for several axes at the same time, one should give a mapping giving the new labels for each changed axis

```

>>> a.set_labels({'sex': 'Men,Women', 'nat': 'Belgian,Foreigner'})
nat\sex | Men | Women

```

(continues on next page)

(continued from previous page)

Belgian		0		1
Foreigner		2		3

one can also replace some labels in several axes by giving a mapping of mappings

```
>>> a.set_labels({'sex': {'M': 'Men'}, 'nat': {'BE': 'Belgian'}})
nat\sex | Men | F
Belgian |   0 | 1
        FO |   2 | 3
```

- allowed matrix multiplication (@ operator) between arrays with dimension != 2 (closes [issue 122](#)).
- improved LArray.plot to get nicer plots by default. The axes are transposed compared to what they used to, because the last axis is often used for time series. Also it considers a 1D array like a single series, not N series of 1 point.
- added installation instructions (closes [issue 101](#)).
- Axis.group and Axis.all are now deprecated (closes [issue 148](#)).

```
>>> city.group(['London', 'Brussels'], name='capitals')
# should be written as:
>>> city[['London', 'Brussels']] >> 'capitals'
```

and

```
>>> city.all()
# should be written as:
>>> city[:] >> 'all'
```

Fixes

- viewer: allow changing the number of displayed digits even for integer arrays as that makes sense when using scientific notation (closes [issue 100](#)).
- viewer: fixed opening a viewer via view() edit() or compare() from within the viewer (closes [issue 109](#))
- viewer: fixed compare() colors when arrays have values which are very close but not exactly equal (closes [issue 123](#))
- viewer: fixed legend when plotting arbitrary rows (it always displayed the labels of the first rows) (closes [issue 136](#)).
- viewer: fixed labels on the x axis when zooming on a plot (closes [issue 143](#))
- viewer: fixed storing an array in a variable with a name which existed previously but which was not displayable in the viewer, such as the name of any function or special object. In some cases, this error lead to a crash of the viewer. For example, this code failed when run in the viewer console, because x is already defined (for the x. syntax):

```
>>> x = ndtest(3)
```

- fixed indexing an array using a positional group with a position which corresponds to a label on that axis. This used to return the wrong data (the data corresponding to the position as if it was the key).

```
>>> a = Axis('a', '1..3')
>>> arr = ndrange(a)
>>> arr
a | 1 | 2 | 3
  | 0 | 1 | 2
>>> # this used to return 0 !
>>> arr[a.i[1]]
1
```

- fixed == for positional groups (closes [issue 93](#))

```
>>> years = Axis('years', '1995..1997')
>>> years
Axis('years', [1995, 1996, 1997])
>>> # this used to return False
>>> years.i[0] == 1995
True
```

- fixed using positional groups for their value in many cases (slice bounds, within list of values, within other groups, etc.). For example, this used to fail:

```
>>> arr = ndtest((2, 4))
>>> arr
a\b | b0 | b1 | b2 | b3
a0 | 0 | 1 | 2 | 3
a1 | 4 | 5 | 6 | 7
>>> b = arr.b
>>> start = b.i[0] # equivalent to start = 'b0'
>>> stop = b.i[2] # equivalent to stop = 'b2'
>>> arr[start:stop]
a\b | b0 | b1 | b2
a0 | 0 | 1 | 2
a1 | 4 | 5 | 6
>>> arr[[b.i[0], b.i[2]]]
a\b | b0 | b2
a0 | 0 | 2
a1 | 4 | 6
```

- fixed posargsort labels (closes [issue 137](#)).
- fixed labels when doing group aggregates using positional groups. Previously, it used the positions as labels. This was most visible when using the `Group.by()` method (which creates positional groups).

```
>>> years = Axis('years', '2010..2015')
>>> arr = ndrange(years)
>>> arr
years | 2010 | 2011 | 2012 | 2013 | 2014 | 2015
      | 0 | 1 | 2 | 3 | 4 | 5
>>> arr.sum(years.by(3))
years | 2010:2012 | 2013:2015
      | 3 | 12
```

While this used to return:

```
>>> arr.sum(years.by(3))
years | 0:3 | 3:6
      | 3 | 12
```

- fixed `Group.by()` when the group was a slice with either bound unspecified. For example, `years[2010:2015].by(3)` worked but `years[:].by(3)`, `years[2010:].by(3)` and `years[:2015].by(3)` did not.
- fixed a speed regression in version 0.18 and later versions compared to 0.17. In some cases, it was up to 40% slower than it should (closes [issue 165](#)).

6.1.17 Version 0.20

Released on 2017-02-09.

IMPORTANT

To make sure all users have all optional dependencies installed and use the same version of packages, and to simplify the update process, we created a new “larrayenv” package which will install larray itself AND all its dependencies (*including* the optional ones). This means that this version needs to be installed using:

```
conda install larrayenv
```

in the future, to update from one version to the next, it should always be enough to do:

```
conda update larrayenv
```

New features

- implemented `from_lists()` to create constant arrays (instead of using `LArray` directly as that is very error prone). We are not really happy with its name though, so it might change in the future. Any suggestion of a better name is very welcome (closes [issue 30](#)).

```
>>> from_lists([[ 'sex\year', 1991, 1992, 1993],
...             [ 'M',          0,    1,    2],
...             [ 'F',          3,    4,    5]])
sex\year | 1991 | 1992 | 1993
      M |    0 |    1 |    2
      F |    3 |    4 |    5
```

- added support for loading sparse arrays via `open_excel()`.

For example, assuming you have a sheet like this:

```
age | sex\year | 2015 | 2016
10 |          F | 0.0 | 1.0
10 |          M | 2.0 | 3.0
20 |          M | 4.0 | 5.0
```

loading it will yield:

```
>>> wb = open_excel('test_sparse.xlsx')
>>> arr = wb['Sheet1'].load()
>>> arr
age | sex\year | 2015 | 2016
10 |          F | 0.0 | 1.0
10 |          M | 2.0 | 3.0
20 |          F | nan | nan
20 |          M | 4.0 | 5.0
```

Miscellaneous improvements

- allowed to get an axis from an array by using `array.axis_name` in addition to `array.axes.axis_name`:

```
>>> arr = ndtest((2, 3))
>>> arr.axes
AxisCollection([
    Axis('a', ['a0', 'a1']),
    Axis('b', ['b0', 'b1', 'b2'])
])
>>> arr.a
Axis('a', ['a0', 'a1'])
```

- viewer: several rows/columns can be plotted together. It draws a separate line for each row except if only one column has been selected.
- viewer: the array labels are used as “ticks” in plots.
- ‘_by’ aggregation methods accept groups in addition to axes (closes [issue 59](#)). It will keep only the mentioned groups and aggregate all other dimensions:

```
>>> arr = ndtest((2, 3, 4))
>>> arr
a | b\c | c0 | c1 | c2 | c3
a0 | b0 | 0 | 1 | 2 | 3
a0 | b1 | 4 | 5 | 6 | 7
a0 | b2 | 8 | 9 | 10 | 11
a1 | b0 | 12 | 13 | 14 | 15
a1 | b1 | 16 | 17 | 18 | 19
a1 | b2 | 20 | 21 | 22 | 23
```

```
>>> arr.sum_by('c0,c1;c1:c3')
c | c0,c1 | c1:c3
  | 126 | 216
```

- viewer: `view()` and `edit()` now accept as argument a path to a file containing arrays.

```
>>> view('myfile.h5')
```

this is a shortcut for:

```
>>> view(Session('myfile.h5'))
```

- `AxisCollection.without` now accepts a single integer position (to exclude an axis by position).

```
>>> a = ndtest((2, 3))
>>> a.axes
AxisCollection([
    Axis('a', ['a0', 'a1']),
    Axis('b', ['b0', 'b1', 'b2'])
])
>>> a.axes.without(0)
AxisCollection([
    Axis('b', ['b0', 'b1', 'b2'])
])
```

- nicer display (`repr`) for `LSet` (closes [issue 44](#)).

```
>>> x.b['b0,b2'].set()
x.b['b0', 'b2'].set()
```

- implemented sep argument for LArray & AxisCollection.combine_axes() to allow using a custom delimiter (closes [issue 53](#)).
- added a check that ipfp target sums have expected axes (closes [issue 42](#)).
- when the nb_index argument is not provided explicitly in read_excel(engine='xlrd'), it is autodetected from the position of the first "" (closes [issue 66](#)).
- allow any special character except "." and whitespace when creating axes labels using "." syntax (previously only _ was allowed).
- added many more I/O tests to hopefully lower our regression rate in the future (closes [issue 70](#)).

Fixes

- viewer: selection of entire rows/columns will load any remaining data, if any (closes [issue 37](#)). Previously if you selected entire rows or columns of a large dataset (which is not loaded entirely from the start), it only selected (and thus copied/plotted) the part of the data which was already loaded.
- viewer: filtering on anonymous axes is now possible (closes [issue 33](#)).
- fixed loading sparse files using read_excel() (fixes [issue 29](#)).
- fixed nb_index argument for read_excel().
- fixed creating range axes with a negative start bound using string notation (e.g. Axis('name', '-1..10')) (fixes [issue 51](#)).
- fixed ptp() function.
- fixed with_axes() to copy the title of the array.
- fixed Group >> 'name'.
- fixed workbook[sheet_position] when using open_excel().
- fixed plotting in the viewer when using Qt4.

6.1.18 Version 0.19

Released on 2017-01-19.

New features

- Implemented a "by" variant to all aggregate methods (e.g. sum_by, mean_by, etc.). These methods aggregate all axes except those listed, which means the only axes remaining after the aggregate operation will be those listed. For example: arr.sum_by(x.a) is equivalent to arr.sum(arr.axes - x.a)

```
>>> arr = ndtest((2, 3, 4))
>>> arr
  a | b\c | c0 | c1 | c2 | c3
a0 | b0 | 0 | 1 | 2 | 3
a0 | b1 | 4 | 5 | 6 | 7
a0 | b2 | 8 | 9 | 10 | 11
a1 | b0 | 12 | 13 | 14 | 15
```

(continues on next page)

(continued from previous page)

```

a1 | b1 | 16 | 17 | 18 | 19
a1 | b2 | 20 | 21 | 22 | 23
>>> arr.sum_by(x.b)
b | b0 | b1 | b2
  | 60 | 92 | 124

```

- Added `.extend()` method to Axis class

```

>>> a = Axis('a', 'a0..a2')
>>> a
Axis('a', ['a0', 'a1', 'a2'])
>>> other = Axis('other', 'a3..a5')
>>> a.extend(other)
Axis('a', ['a0', 'a1', 'a2', 'a3', 'a4', 'a5'])

```

or directly specify the extra labels as a list or as a “label string”:

```

>>> a.extend('a3..a5')
Axis('a', ['a0', 'a1', 'a2', 'a3', 'a4', 'a5'])

```

- Added title argument to all array creation functions (`ndrange`, `zeros`, `ones`, ...) and display it in the `.info` of array objects.

```

>>> a = ndrange(3, title='a simple test array')
>>> a.info
a simple test array
3
{0}* [3]: 0 1 2

```

- implemented creating an Axis using a group:

```

>>> arr = ndtest((2, 3))
>>> arr
a\b | b0 | b1 | b2
a0 | 0 | 1 | 2
a1 | 3 | 4 | 5
>>> a, b = arr.axes
>>> zeros((a, b['b1']))
a\b | b0 | b1
a0 | 0.0 | 0.0
a1 | 0.0 | 0.0

```

- made `Axis.startswith`, `.endswith` and `.matches` accept Group instances

```

>>> a = Axis('a', 'a0..b2')
>>> a
Axis('a', ['a0', 'a1', 'a2', 'b0', 'b1', 'b2'])

```

```

>>> prefix = Axis('prefix', 'a,b')
>>> a.startswith(prefix['a'])
a['a0', 'a1', 'a2']
>>> a.startswith(prefix.i[1])
a['b0', 'b1', 'b2']

```

- implemented all usual binary operations (+, -, *, /, ...) on Group

```
>>> year = Axis('year', '2011..2016')
>>> year[2013] + 1
2014
>>> year.i[2] + 1
2014
```

- made the viewer is much more useful as a debugger in the middle of a function by generalizing SessionEditor to handle any mapping, instead of only Session objects but made it list and display only array objects. To view the value of non-array variable one should type their name in the console. Given those changes, view() will superficially behave as before, but behind the scene, *all* variables which were defined in the scope where view() was called will be available in the viewer console, even though they will not appear in the list on the left. This means that the viewer console will be able to use scalars defined at that point and call others functions of your code. In other words, there are more chances you can execute some code from the function calling view() by simply copy-pasting the code line.

Backward incompatible changes

- LGroup lost set-like operations (intersection and union) to the profit of a specific subclass (LSet). In other words, this no longer works:

```
>>> letters = Axis('letters', 'a..z')
>>> letters[':c'] & letters['b:']
```

To make it work, we need to convert the LGroup(s) to LSets explicitly:

```
>>> letters[':c'].set() & letters['b:d'].set()
letters.set[OrderedSet(['b', 'c'])]
```

```
>>> letters[':c'].set() | letters['b:d'].set()
letters.set[OrderedSet(['a', 'b', 'c', 'd'])]
```

```
>>> letters[':c'].set() - 'b'
letters.set[OrderedSet(['a', 'c'])]
```

- group aggregates produce simple string labels for the new aggregated axis instead of using the group themselves as labels. This means one can no longer know where a group comes from but this simplifies the code and fixes a few issues, most notably export of aggregated arrays to Excel, and some operations between two aggregated arrays.

```
>>> arr = ndtest((3, 4))
>>> arr
a\b | b0 | b1 | b2 | b3
a0 | 0 | 1 | 2 | 3
a1 | 4 | 5 | 6 | 7
a2 | 8 | 9 | 10 | 11
>>> agg = arr.sum(':b2 >> tob2;b2,b3 >> other')
>>> agg
a\b | tob2 | other
a0 | 3 | 5
a1 | 15 | 13
a2 | 27 | 21
>>> agg.info
3 x 2
a [3]: 'a0' 'a1' 'a2'
```

(continues on next page)

(continued from previous page)

```
b [2]: 'tob2' 'other'
>>> agg.axes.b.labels[0]
'tob2'
```

In previous versions this would have returned:

```
>>> agg.axes.b.labels[0]
LGroup('b2', name='tob2', axis=Axis('b', ['b0', 'b1', 'b2', 'b3']))
```

- a string containing only a single “integer-like” is no longer transformed to an integer e.g. “10” will evaluate to (the string) “10” (like in version 0.17 and earlier) while “10,20” will evaluate to the list of integers: [10, 20]

Other changes

- changed how Group instances are displayed.

```
>>> a = Axis('a', 'a0..a2')
>>> a['a1,a2']
a['a1', 'a2']
```

Fixes

- fixed > and >= on Group using slices
- avoid a division by 0 warning when using divnot0
- viewer: fixed plots when Qt5 is installed. This also removes the matplotlib warning people got when running the viewer with Qt5 installed.
- viewer: display array when typing its name in the console even when no array was selected previously

Misc

- misc code cleanup, improved docstrings, ...

6.1.19 Version 0.18

Released on 2016-12-20.

Major improvements

- the documentation (docstrings) of many functions was vastly improved (thanks to Alix)
- implemented a new optional syntax to generate sequences of labels for axes by using patterns
integer strings generate integers

```
>>> ndrange('age=0..10')
age | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

you can combine letters and numbers. The number part is treated like increasing (or decreasing numbers)

```
>>> ndrange('lipro=P01..P12')
lipro | P01 | P02 | P03 | P04 | P05 | P06 | P07 | P08 | P09 | P10 | P11 | P12
      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11
```

letter patterns generate all combination of letters between the start and end:

```
>>> ndrange('test=AA..CC')
test  | AA | AB | AC | BA | BB | BC | CA | CB | CC
      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
```

other characters are left intact (and should be the same on the start and end patterns):

```
>>> ndrange('test=A_1..C_2')
test  | A_1 | A_2 | B_1 | B_2 | C_1 | C_2
      | 0  | 1  | 2  | 3  | 4  | 5
```

this also works within Axis()

```
>>> Axis('age', '0..10')
Axis('age', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

- implemented new syntax for defining groups using strings:

```
>>> arr = ndtest((3, 4))
>>> arr
a\b | b0 | b1 | b2 | b3
a0 | 0  | 1  | 2  | 3
a1 | 4  | 5  | 6  | 7
a2 | 8  | 9  | 10 | 11
```

groups can be named using “>>” instead of “=” previously

```
>>> arr.sum('b1,b3 >> b13;b0:b2 >> b012')
a\b | b13 | b012
a0 | 4  | 3
a1 | 12 | 15
a2 | 20 | 27
```

if some labels are ambiguous, one can specify the axis by using “axis_name[labels]”:

```
>>> arr.sum('b[b1,b3] >> b13;b[b0:b2] >> b012')
a\b | b13 | b012
a0 | 4  | 3
a1 | 12 | 15
a2 | 20 | 27
```

groups can also be defined by position using this syntax:

```
>>> arr.sum('b.i[1,3] >> b13;b.i[0:3] >> b012')
a\b | b13 | b012
a0 | 4  | 3
a1 | 12 | 15
a2 | 20 | 27
```

A few notes:

- the goal was to have that syntax as close as the “normal” syntax as possible (just remove the “x.” and all inner quotes).

- in models, the normal syntax should be preferred, so that the groups can be stored in a variable and reused in several places
- strings representing integers are evaluated as integers.
- there is experimental support for evaluating expressions within string groups by using “{expr}”, but this is fragile and might be removed in the future.
- implemented `combine_axes` & `split_axis` on arrays:

```
>>> arr = ndtest((2, 3, 4))
>>> arr
a | b\c | c0 | c1 | c2 | c3
a0 | b0 | 0 | 1 | 2 | 3
a0 | b1 | 4 | 5 | 6 | 7
a0 | b2 | 8 | 9 | 10 | 11
a1 | b0 | 12 | 13 | 14 | 15
a1 | b1 | 16 | 17 | 18 | 19
a1 | b2 | 20 | 21 | 22 | 23
```

```
>>> arr2 = arr.combine_axes((x.a, x.b))
>>> arr2
a_b\c | c0 | c1 | c2 | c3
a0_b0 | 0 | 1 | 2 | 3
a0_b1 | 4 | 5 | 6 | 7
a0_b2 | 8 | 9 | 10 | 11
a1_b0 | 12 | 13 | 14 | 15
a1_b1 | 16 | 17 | 18 | 19
a1_b2 | 20 | 21 | 22 | 23
```

```
>>> arr2.split_axis(x.a_b)
a | b\c | c0 | c1 | c2 | c3
a0 | b0 | 0 | 1 | 2 | 3
a0 | b1 | 4 | 5 | 6 | 7
a0 | b2 | 8 | 9 | 10 | 11
a1 | b0 | 12 | 13 | 14 | 15
a1 | b1 | 16 | 17 | 18 | 19
a1 | b2 | 20 | 21 | 22 | 23
```

- implemented `.by()` method on groups which splits them into subgroups of specified length

```
>>> arr = ndtest((5, 2))
>>> arr
a\b | b0 | b1
a0 | 0 | 1
a1 | 2 | 3
a2 | 4 | 5
a3 | 6 | 7
a4 | 8 | 9
```

```
>>> arr.sum(a['a0': 'a4'].by(2))
a\b | b0 | b1
a['a0' 'a1'] | 2 | 4
a['a2' 'a3'] | 10 | 12
a['a4'] | 8 | 9
```

there is also an optional second argument to specify the “step” between groups

```
>>> arr.sum(a['a0':'a4'].by(2, step=3))
      a\b | b0 | b1
a['a0' 'a1'] | 2 | 4
a['a3' 'a4'] | 14 | 16
```

if the step is < the group size, you get overlapping groups:

```
>>> arr.sum(a['a0':'a4'].by(2, step=1))
      a\b | b0 | b1
a['a0' 'a1'] | 2 | 4
a['a1' 'a2'] | 6 | 8
a['a2' 'a3'] | 10 | 12
a['a3' 'a4'] | 14 | 16
      a['a4'] | 8 | 9
```

- groups can be renamed using >> (in addition to the “named” method)

```
>>> arr = ndtest((2, 3))
>>> arr
a\b | b0 | b1 | b2
a0 | 0 | 1 | 2
a1 | 3 | 4 | 5
>>> arr.sum((x.b['b0,b1'] >> 'b01', x.b['b1,b2'] >> 'b12'))
a\b | b01 | b12
a0 | 1 | 3
a1 | 7 | 9
```

- implemented `rationot0`

```
>>> a = Axis('a', 'a0,a1')
>>> b = Axis('b', 'b0,b1,b2')
>>> arr = LArray([[6, 0, 2],
...               [4, 0, 8]], [a, b])
>>> arr
a\b | b0 | b1 | b2
a0 | 6 | 0 | 2
a1 | 4 | 0 | 8
>>> arr.sum()
20
>>> arr.rationot0()
a\b | b0 | b1 | b2
a0 | 0.3 | 0.0 | 0.1
a1 | 0.2 | 0.0 | 0.4
>>> arr.rationot0(x.a)
a\b | b0 | b1 | b2
a0 | 0.6 | 0.0 | 0.2
a1 | 0.4 | 0.0 | 0.8
```

for reference, the normal ratio method would return:

```
>>> arr.ratio(x.a)
a\b | b0 | b1 | b2
a0 | 0.6 | nan | 0.2
a1 | 0.4 | nan | 0.8
```

Misc improvements

- implemented `[]` on groups so that you can further subset them
- added a new “condensed” option for `ipfp`’s `display_progress` argument to get back the old behavior
- changed how named groups are displayed (only the name is displayed)
- positional groups gained a few features and are almost on par with label groups now
- when iterating over an axis (for example when doing “for y in year_axis:” it yields groups (instead of raw labels) so that it works even in the presence of ambiguous labels.
- `Axis.startswith`, `endswith`, `matches` create groups which include the axis (so that those groups work even if the labels exist on several axes)

Bug fixes

- fixed `Session.summary()` when arrays in the session have axes without name
- fixed `full()` and `full_like()` with an explicit dtype (the dtype was ignored)

6.1.20 Version 0.17

Released on 2016-11-29.

Core

- added `ndtest` function to create n-dimensional test arrays (of given shape). Axes are named by single letters starting from ‘a’. Axes labels are constructed using a ‘{axis_name}{label_pos}’ pattern (e.g. ‘a0’).

```
>>> ndtest(6)
a | a0 | a1 | a2 | a3 | a4 | a5
  | 0 | 1 | 2 | 3 | 4 | 5
>>> ndtest((2, 3))
a\b | b0 | b1 | b2
a0 | 0 | 1 | 2
a1 | 3 | 4 | 5
>>> ndtest((2, 3), label_start=1)
a\b | b1 | b2 | b3
a1 | 0 | 1 | 2
a2 | 3 | 4 | 5
```

- allow naming “one-shot” groups in group aggregates.

```
>>> arr = ndtest((2, 3))
>>> arr
a\b | b0 | b1 | b2
a0 | 0 | 1 | 2
a1 | 3 | 4 | 5
>>> arr.sum('g1=b0;g2=b1,b2;g3=b0:b2')
a\b | 'g1' ('b0') | 'g2' (['b1' 'b2']) | 'g3' ('b0':'b2')
a0 | 0 | 3 | 3
a1 | 3 | 9 | 12
```

- implemented `argmin`, `argmax`, `posargmin`, `posargmax` without an axis argument (works on the full array).

```
>>> arr = ndtest((2, 3))
>>> arr
a\b | b0 | b1 | b2
a0 |  0 |  1 |  2
a1 |  3 |  4 |  5
>>> arr.argmin()
('a0', 'b0')
```

- added preliminary code to add a title attribute to LArray.

This needs a lot more work to be really useful though, as it can currently only be used in the LArray() function itself and is only used in Session.summary() (see below). There are many places where this should be used, but this is not done yet.

- added Session.summary() which displays a list of all arrays, their dimension names and title if any.

This can be used in combination with local_arrays() to produce some kind of codebook with all the arrays of a function.

```
>>> arr = LArray([[1, 2], [3, 4]], 'sex=M,F;nat=BE,FO', title='a test array')
>>> arr
sex\nat | BE | FO
      M |  1 |  2
      F |  3 |  4
>>> s = Session({'arr': arr})
>>> s
Session(arr)
>>> print(s.summary())
arr: sex, nat
     a test array
```

- fixed using groups from other (compatible) axis
- fixed group aggregates using groups without axis
- fixed axis[another_label_group] when said group had a non-string Axis
- fixed axis.group(another_label_group, name='a_name') (name was not set correctly)
- fixed ipfp progress message when progress is negative

viewer

- when setting part of an array in the console (by using e.g. arr['M'] = 10), display that array
- when typing in the console the name of an existing array, select it in the list
- fixed missing tooltips for arrays added to the session from within the session viewer
- fixed window title (with axes info) not updating in many cases
- fixed the filters bar not being cleared when displaying a non-LArray object after an LArray object

misc

- improved messages in `ipfp(display_progress=True)`
- improved tests, docstrings, ...

6.1.21 Version 0.16.1

Released on 2016-11-04.

Viewer

- renamed “Ok” button in array/session viewer to “Close”.
- added apply and discard buttons in session editor, which permanently apply or discard changes to the current array.

Core

- fixed `array[sequence, scalar] = value`
- fixed `array.to_excel()` which was broken in 0.16 (by the upgrade to `xlwings` 0.9+).
- improved a few tests

6.1.22 Version 0.16

Released on 2016-10-26.

Warning: this release needs to be installed using:

```
conda update larray conda update xlwings
```

New features

- implemented support for `xlwings` 0.9+. This allowed us to change the way we interact with Excel:
 - by default, the Excel instance we use is configured to be both hidden and silent (for example, it does not prompt to update/edit links).
 - by default, we now use a dedicated Excel instance for each call to `open_excel`, instead of reusing any existing instance if there was any open. In practice, it means input/output from/to Excel is more reliable and does not risk altering any workbook you had open (except if you ask for that explicitly). The cost of this is that it is slower by default. If you open many different workbooks, it is recommended that you create a single Excel instance and reuse it. This can be done with:

```
>>> from larray import *
>>> import xlwings as xw

>>> app = xw.App(visible=False, add_book=False)
>>> wb1 = open_excel('workbook1.xlsx', app=app)
# use wb1 as before
>>> wb1.close()
>>> wb2 = open_excel('workbook2.xlsx', app=app)
```

(continues on next page)

(continued from previous page)

```
# use wb2 as before
>>> wb2.close()
>>> app.quit()
```

- added `ipfp` function which does Iterative Proportional Fitting Procedure (also known as bi-proportional fitting in statistics or RAS algorithm in economics). Note that this new function is currently not in the core module, so it needs a specific import command:

```
>>> from larray.ipfp import ipfp
```

```
>>> a = Axis('a', 2)
>>> b = Axis('b', 2)
>>> initial = LArray([[2, 1],
...                  [1, 2]], [a, b])
>>> initial
a\b* | 0 | 1
    0 | 2 | 1
    1 | 1 | 2
>>> target_sum_along_a = LArray([2, 1], b)
>>> target_sum_along_b = LArray([1, 2], a)
>>> ipfp([target_sum_along_a, target_sum_along_b], initial, threshold=0.01)
a\b* |      0 |      1
    0 | 0.8450704225352113 | 0.15492957746478875
    1 | 1.1538461538461537 | 0.8461538461538463
```

- made it possible to create arrays more succinctly in some usual cases (especially for quick arrays for testing purposes). Previously, when one created an array from scratch, he had to provide Axis object(s) (or another array). Note that the following examples use `zeros()` but this change affects all array creation functions (`ones`, `zeros`, `ndrange`, `full`, `empty`):

```
>>> nat = Axis('nat', ['BE', 'FO'])
>>> sex = Axis('sex', ['M', 'F'])
>>> zeros([nat, sex])
nat\sex |  M |  F
      BE | 0.0 | 0.0
      FO | 0.0 | 0.0
```

Now, when you have axe names and axes labels but do not have/want to reuse an existing axis, you can use this syntax:

```
>>> zeros([('nat', ['BE', 'FO']),
...       ('sex', ['M', 'F'])])
nat\sex |  M |  F
      BE | 0.0 | 0.0
      FO | 0.0 | 0.0
```

If additionally all axe names and labels are strings (not integers or other types) which do not contain any special character (“=”, “,” or “;”) you can use:

```
>>> zeros('nat=BE,FO;sex=M,F')
nat\sex |  M |  F
      BE | 0.0 | 0.0
      FO | 0.0 | 0.0
```

See below (*) for some more alternate syntaxes and an explanation of how this works.

- added additional, less error-prone syntax for stack:

```

>>> nat = Axis('nat', 'BE,FO')
>>> arr1 = ones(nat)
>>> arr1
nat | BE | FO
    | 1.0 | 1.0
>>> arr2 = zeros(nat)
>>> arr2
nat | BE | FO
    | 0.0 | 0.0
>>> stack([('M', arr1), ('F', arr2)], 'sex')
nat\sex | H | F
        BE | 1.0 | 0.0
        FO | 1.0 | 0.0

```

in addition to the still supported but discouraged (because one has to remember the order of labels):

```

>>> sex = Axis('sex', ['M', 'F'])
>>> stack((arr1, arr2), sex)
nat\sex | H | F
        BE | 1.0 | 0.0
        FO | 1.0 | 0.0

```

- added `LArray.compact` and `Session.compact()` to detect and remove “useless” axes (ie axes for which values are constant over the whole axis)

```

>>> a = LArray([[1, 2], [1, 2]], [Axis('sex', 'M,F'), Axis('nat', 'BE,FO')])
>>> a
sex\nat | BE | FO
        M | 1 | 2
        F | 1 | 2
>>> a.compact()
nat | BE | FO
    | 1 | 2

```

- made `Session` keep the order in which arrays were added to it. The main goal was to make this work:

```

>>> b, a = s['b', 'a']

```

Previously, since sessions were always traversed alphabetically, this was a dangerous operation because if the keys (a and b) were not sorted alphabetically, the result would not be in the expected order:

`s['b', 'a']` previously returned a, b instead of b, a !!

`Session.names` is still sorted alphabetically though (`Session.keys()` is not)

- added `LArray.with_axes(axes)` to return a new `LArray` with the same data but different axes

```

>>> a = ndrange(2)
>>> a
{0}* | 0 | 1
      | 0 | 1
>>> a.with_axes([Axis('sex', 'H,F')])
sex | H | F
    | 0 | 1

```

- changed width from which an `LArray` is summarized (using “...”) from 80 characters to 200.
- implemented `memory_used` property which displays nbytes in human-readable form

```
>>> a = ndrange('sex=H,F;nat=BE,FO')
>>> a.memory_used
'16 bytes'
>>> a = ndrange(100000)
>>> a.memory_used
'390.62 Kb'
```

- implemented Axis + AxisCollection

```
>>> a = ndrange('sex=M,F;type=t1,t2')
>>> Axis('nat', 'BE,FO') + a.axes
AxisCollection([
    Axis('nat', ['BE', 'FO']),
    Axis('sex', ['M', 'F']),
    Axis('type', ['t1', 't2'])
])
```

(*) For the curious, there are also many syntaxes supported for array creation functions. In fact, during array creation, at any place a list or tuple of values is expected, you can specify it using a single string, which will be split successively at the following characters if present: “;” then “=” then “,”. If you apply that algorithm to ‘nat=BE,FO;sex=M,F’, you get:

- 1) ‘nat=BE,FO;sex=M,F’
- 2) (‘nat=BE,FO’, ‘sex=M,F’)
- 3) ((‘nat’, ‘BE,FO’), (‘sex’, ‘M,F’))
- 4) ((‘nat’, (‘BE’, ‘FO’)), (‘sex’, (‘M’, ‘F’)))

Recognise this last syntax? This is the same as above, except above we replaced some () with [] for clarity. In fact all the intermediate forms here above are valid (and equivalent) in array creation functions.

6.1.23 Version 0.15

Released on 2016-09-23.

Core

- added new methods on axes: matches, startswith, endswith

```
>>> country = Axis('country', ['FR', 'BE', 'DE', 'BR'])
>>> country.matches('BE|FR')
LGroup(['FR', 'BE'])
>>> country.matches('^..$') # labels 2 characters long
LGroup(['FR', 'BE', 'DE', 'BR'])
```

```
>>> country.startswith('B')
LGroup(['BE', 'BR'])
>>> country.endswith('R')
LGroup(['FR', 'BR'])
```

- implemented set-like operations on LGroup: & (intersection), | (union), - (difference). Slice groups do not work yet on axes references (x.) but that will come in the future...

```
>>> alpha = Axis('alpha', 'a,b,c,d')
>>> alpha['a', 'b'] | alpha['c', 'd']
LGroup(['a', 'b', 'c', 'd'], axis=...)
>>> alpha['a', 'b', 'c'] | alpha['c', 'd']
LGroup(['a', 'b', 'c', 'd'], axis=...)
```

a name is computed automatically when both operands are named

```
>>> r = alpha['a', 'b'].named('ab') | alpha['c', 'd'].named('cd')
>>> r.name
'ab | cd'
>>> r.key
['a', 'b', 'c', 'd']
```

numeric axes work too

```
>>> num = Axis('num', range(10))
>>> num[:2] | num[8:]
num[0, 1, 2, 8, 9]
>>> num[:2] | num[5]
num[0, 1, 2, 5])
```

intersection

```
>>> LGroup(['a', 'b', 'c']) & LGroup(['c', 'd'])
LGroup(['c'])
```

difference

```
>>> LGroup(['a', 'b', 'c']) - LGroup(['c', 'd'])
LGroup(['a', 'b'])
>>> LGroup(['a', 'b', 'c']) - 'b'
LGroup(['a', 'c'])
```

- fixed loading 1D arrays using open_excel

Viewer

- added tooltip with the axes labels corresponding to each cell of the array viewer
- added name and dimensions of the current array to the window title bar in the session viewer
- added tooltip with each array .info() in the list of arrays of the session viewer
- fixed eval box throwing an exception when trying to set a new variable (if qtconsole is not present)
- fixed group aggregates using LGroups defined using axes references (x.), for example:

```
>>> arr.sum(x.age[:10])
```

- fixed group aggregates using anonymous axes

6.1.24 Version 0.14.1

Released on 2016-08-12.

Fixes

- fixed support for loading arrays without axe names from Excel files (in that case `index_col/nb_index` are necessary)
- fixed using a single int for `index_col` in `read_excel()` and `sheet.load()`
- fixed loading empty Excel sheets via `xlwings` correctly (ie do not crash)
- fixed dumping a session loaded from an H5 file to Excel

6.1.25 Version 0.14

Released on 2016-08-10.

Important warning

This version is not compatible with the new version of `xlwings` that just came out. Consequently, upgrading to this version is different from the usual “conda update larray”. You should rather use:

```
conda update larray --no-update-deps
```

To get the most of this release, you should also install the “qtconsole” package via:

```
conda install qtconsole
```

Viewer

- upgraded session viewer/editor to work like a super-calculator. The input box below the array view can be used to type any expression. eg `array1.sum(x.age) / array2`, which will be displayed in the viewer. One can also type assignment commands, like: `array3 = array1.sum(x.age) / array2` In which case, the new array will be displayed in the viewer AND added to the session (appear on the list on the left), so that you can use it in other expressions.

If you have the “qtconsole” package installed (see above), that input box will be a full ipython console. This means:

- history of typed commands,
- tab-completion (for example, type “nd<tab>” and it will change to “ndrange”),
- syntax highlighting,
- calltips (show the documentation of functions when typing commands using them),
- help on functions using “?”. For example, type “ndrange?<enter>” to get the full documentation about `ndrange`. Use <ESC> or <q> to quit that screen !),
- etc.

When having the “qtconsole” package installed, you might get a warning when starting the viewer:

```
WARNING:root:Message signing is disabled. This is insecure and not recommended!
```

This is totally harmless and can be safely ignored !

- made `view()` and `edit()` without argument equivalent to `view(local_arrays())` and `edit(local_arrays())` respectively.
- made the viewer on large arrays start a lot faster by using a small subset of the array to guess the number of decimals to display and whether or not to use scientific notation.
- **improved `compare()`:**
 - added support for comparing sessions. Arrays with differences between sessions are colored in red.
 - use a single array widget instead of 3. This is done by stacking arrays together to create a new dimension. This has the following advantages:
 - * the filter and scrollbars are de-facto automatically synchronized.
 - * any number of arrays can be compared, not just 2. All arrays are compared to the first one.
 - * arrays with different sets of compatible axes can be compared (eg compare an array with its mean along an axis).
 - added label to show maximum absolute difference.
- implemented `edit(session)` in addition to `view(session)`.

Excel support

- added support for copying sheets via: `wb['x'] = wb['y']` if 'x' sheet already existed, it is completely overwritten.

Core

- improved performance. My test models run about 10% faster than with 0.13.
- made `cumsum` and `cumprod` aggregate on the last axis by default so that the axis does not need to be specified when there is only one.
- implemented much better support for operations using arrays of different types. For example,
 - fixed `create_sequential` when `mult`, `inc` and `initial` are of different types eg `create_sequential(..., initial=1, inc=0.1)` had an unexpected integer result because it always used the type of the initial value for the output
 - when appending a string label to an integer axis (eg adding total to an age axis by using `with_total()`), the resulting axis should have a mixed type, and not be suddenly all string.
 - `stack()` now supports arrays with different types.
- made `stack` support arrays with different axes (the result has the union of all axes)

For completeness

- use `xlwings` (ie live Excel instance) by default for all Excel input/output, including `read_excel()`, `session.dump` and `session.load/Session(filename)`. This has the advantage of more coherent results among the different ways to load/save data to Excel and that simple sessions correctly survive a round-trip to an `.xlsx` workbook (ie (named) axes are detected properly). However, given the very different library involved, we loose most options that `read_excel` used to provide (courtesy of `pandas.read_excel`) and some bugs were probably introduced in the conversion.
- fixed creating a new file via `open_excel()`
- fixed loading 1D arrays (ranges with height 1 or width 1) via `open_excel()`
- fixed `sheet['A1'] = array` in some cases

- `wb.close()` only really close if the workbook was not already open in Excel when `open_excel` was called (so that we do not close a workbook a user is actually viewing).
- added support for `wb.save(filename)`, or actually for using any relative path, instead of a full absolute path.
- when dumping a session to Excel, sort sheets alphabetically instead of dumping them in a “random” order.
- try to convert float to int in more situations
- added support for using `stack()` without providing an axis. It creates an anonymous wildcard axis of the correct length.
- added `aslarray()` top-level function to translate anything into an LArray if it is not already one
- made `labels_array` available via *from larray import **
- fixed binary operations between an array and an axis where the array appeared first (eg `array > axis`). Confusingly, `axis < array` already worked.
- added check in `“a[bool_larray_key]”` to make sure `key.axes` are compatible with `a.axes`
- made `create_sequential` a lot faster when `mult` or `inc` are constants
- made axes without name compatible with any name (this is the equivalent of a wildcard name for labels)
- misc cleanup/docstring improvements/improved tests/improved error messages

6.1.26 Version 0.13

Released on 2016-07-11.

New features

- implemented a new way to do input/output from/to Excel

```
>>> a = ndrange((2, 3))
>>> wb = open_excel('c:/tmp/y.xlsx')
# put a at A1 in Sheet1, excluding headers (labels)
>>> wb['Sheet1'] = a
# dump a at A1 in Sheet2, including headers (labels)
>>> wb['Sheet2'] = a.dump()
# save the file to disk
>>> wb.save()
# close it
>>> wb.close()
```

```
>>> wb = open_excel('c:/tmp/y.xlsx')
# load a from the data starting at A1 in Sheet1, assuming the absence of headers.
>>> a1 = wb['Sheet1']
# load a from the data starting at A1 in Sheet1, assuming the presence of
↳ (correctly formatted) headers.
>>> a2 = wb['Sheet2'].load()
>>> wb.close()
```

```
>>> wb = open_excel('c:/tmp/y.xlsx')
# note that Sheet2 must exist
>>> sheet2 = wb['Sheet2']
# write a without labels starting at C5
```

(continues on next page)

(continued from previous page)

```
>>> sheet2['C5'] = a
# write a with its labels starting at A10
>>> sheet2['A10'] = a.dump()
```

load an array with its axes information from a range. As you might have guessed, we could also use the `sheet2` variable here

```
>>> b = wb['Sheet2']['A10:D12'].load()
>>> b
{0}* \ {1}* | 0 | 1 | 2
           0 | 0 | 1 | 2
           1 | 3 | 4 | 5
```

load an array (raw data) with no axis information from a range.

```
>>> c = sheet['B11:D12']
>>> # in fact, this is not really an LArray ...
>>> c
<larray.excel.Range at 0xlff1bae22e8>
>>> # but it can be used as such (this is currently very experimental)
>>> c.sum(axis=0)
{0}* | 0 | 1 | 2
      | 3.0 | 5.0 | 7.0
>>> # ... and it can be used for other stuff, like setting the formula instead of
↳ the value:
>>> c.formula = '=D10+1'
>>> # in the future, we should also be able to set font name, size, style, etc.
```

- implemented `LArray.rename({axis: new_name})` as well as using kwargs to rename several axes at once

```
>>> nat = Axis('nat', ['BE', 'FO'])
>>> sex = Axis('sex', ['M', 'F'])
>>> a = ndrange([nat, sex])
>>> a
nat\sex | M | F
        BE | 0 | 1
        FO | 2 | 3
>>> a.rename(nat='nat2', sex='gender')
nat2\gender | M | F
            BE | 0 | 1
            FO | 2 | 3
>>> a.rename({'nat': 'nat2', 'sex': 'gender'})
nat2\gender | M | F
            BE | 0 | 1
            FO | 2 | 3
```

- made tab-completion of axes names possible in an interactive console

For completeness

- taking a subset of an array with wildcard axes now returns an array with wildcard axes
- fixed a case where wildcard axes were considered incompatible when they actually were compatible
- better support for anonymous axes
- fix for obscure bugs, better doctests, cleaner implementation for a few functions, ...

6.1.27 Version 0.12

Released on 2016-06-21.

New features

- implemented boolean indexing by using axes objects:

```
>>> sex = Axis('sex', 'M,F')
>>> age = Axis('age', range(5))
>>> a = ndrange((sex, age))
>>> a
sex\age | 0 | 1 | 2 | 3 | 4
      M | 0 | 1 | 2 | 3 | 4
      F | 5 | 6 | 7 | 8 | 9
```

```
>>> a[age < 3]
sex\age | 0 | 1 | 2
      M | 0 | 1 | 2
      F | 5 | 6 | 7
```

This new syntax is equivalent to (but currently much slower than):

```
>>> a[age[:2]]
sex\age | 0 | 1 | 2
      M | 0 | 1 | 2
      F | 5 | 6 | 7
```

However, the power of this new syntax comes from the fact that you are not limited to scalar constants

```
>>> age_limit = LArray([2, 3], sex)
>>> age_limit
sex | M | F
    | 2 | 3
```

```
>>> a[age < age_limit]
sex,age | M,0 | M,1 | F,0 | F,1 | F,2
        | 0 | 1 | 5 | 6 | 7
```

Notice that the concerned axes are merged, so you cannot do much as much with them. For example, `a[age < age_limit].sum(x.age)` would not work since there is no “age” axis anymore.

To keep axes intact, one can often set the values of the corresponding cells to 0 or nan instead.

```

>>> a[age < age_limit] = 0
>>> a
sex\age | 0 | 1 | 2 | 3 | 4
        M | 0 | 0 | 2 | 3 | 4
        F | 0 | 0 | 0 | 8 | 9
>>> # in this case, the sum is valid (but the mean would not -- one should use
↳nan for that)
>>> a.sum(x.age)
sex | M | F
    | 9 | 17

```

To keep axes intact, this idiom is also often useful:

```

>>> b = a * (age >= age_limit)
>>> b
sex\age | 0 | 1 | 2 | 3 | 4
        M | 0 | 0 | 2 | 3 | 4
        F | 0 | 0 | 0 | 8 | 9

```

This also works with axes references (`x.axis_name`), though this is experimental and the filter value is only computed as late as possible (during `[]`), so you cannot display it before that, like you can with “real” axes.

Using “real” axes:

```

>>> filter1 = age < age_limit
>>> filter1
age\sex |      M |      F
        0 | True | True
        1 | True | True
        2 | False| True
        3 | False| False
        4 | False| False
>>> a[filter1]
sex,age | M,0 | M,1 | F,0 | F,1 | F,2
        | 0 | 1 | 5 | 6 | 7

```

With axes references:

```

>>> filter2 = x.age < age_limit
>>> filter2
<larray.core.BinaryOp at 0x1332ae3b588>
>>> a[filter2]
sex,age | M,0 | M,1 | F,0 | F,1 | F,2
        | 0 | 1 | 5 | 6 | 7
>>> a * ~filter2
sex\age | 0 | 1 | 2 | 3 | 4
        M | 0 | 0 | 2 | 3 | 4
        F | 0 | 0 | 0 | 8 | 9

```

- implemented `LArray.divnot0`

```

>>> nat = Axis('nat', ['BE', 'FO'])
>>> sex = Axis('sex', ['M', 'F'])
>>> a = ndrange((nat, sex))
>>> a
nat\sex | M | F
        BE | 0 | 1

```

(continues on next page)

(continued from previous page)

```

      FO | 2 | 3
>>> b = ndrange(sex)
>>> b
sex | M | F
    | 0 | 1
>>> a / b
nat\sex | M | F
        BE | nan | 1.0
        FO | inf | 3.0
>>> a.divnot0(b)
nat\sex | M | F
        BE | 0.0 | 1.0
        FO | 0.0 | 3.0

```

- implemented `.named()` on groups to name groups after the fact

```

>>> a = ndrange(Axis('age', range(100)))
>>> a
age | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99
>>> a.sum((x.age[10:19].named('teens'), x.age[20:29].named('twenties')))
age | 'teens' (10:19) | 'twenties' (20:29)
    |                | 145 | 245

```

- made all array creation functions (`ndrange`, `zeros`, `ones`, `full`, `LArray`, ...) more flexible:

They accept a single `Axis` argument instead of requiring a tuple/list of them

```

>>> sex = Axis('sex', 'M,F')
>>> a = ndrange(sex)
>>> a
sex | M | F
    | 0 | 1

```

Shortcut definition for axes work

```

>>> ndrange("a,b,c")
{0} | a | b | c
    | 0 | 1 | 2
>>> ndrange(["1:3", "d,e"])
{0}\{1} | d | e
        1 | 0 | 1
        2 | 2 | 3
        3 | 4 | 5
>>> LArray([1, 5, 7], "a,b,c")
{0} | a | b | c
    | 1 | 5 | 7

```

One can mix `Axis` objects and ints (for axes without labels)

```

>>> sex = Axis('sex', 'M,F')
>>> ndrange([sex, 3])
sex\{1}* | 0 | 1 | 2
         M | 0 | 1 | 2
         F | 3 | 4 | 5

```

- made it possible to iterate on labels of a group (eg a slice of an `Axis`):

```
>>> for year in a.axes.year[2010:]:
...     # do stuff
```

- changed representation of anonymous axes from “axisN” (where N is the position of the axis) to “{N}”. The problem was that “axisN” was not recognizable enough as an anonymous axis, and it was thus misleading. For example “a[x.axis0[...]]” would not work.
- better overall support for arrays with anonymous axes or several axes with the same name
- fixed all output functions (to_csv, to_excel, to_hdf, ...) when the last axis has no name but other axes have one
- implemented eye() which creates 2D arrays with ones on the diagonal and zeros elsewhere.

```
>>> eye(sex)
sex\sex |    M |    F
      M | 1.0 | 0.0
      F | 0.0 | 1.0
```

- implemented the @ operator to do matrix multiplication (Python3.5+ only)
- implemented inverse() to return the (matrix) inverse of a (square) 2D array

```
>>> a = eye(sex) * 2
>>> a
sex\sex |    M |    F
      M | 2.0 | 0.0
      F | 0.0 | 2.0
```

```
>>> a @ inverse(a)
sex\sex |    M |    F
      M | 1.0 | 0.0
      F | 0.0 | 1.0
```

- implemented diag() to extract a diagonal or construct a diagonal array.

```
>>> nat = Axis('nat', ['BE', 'FO'])
>>> sex = Axis('sex', ['M', 'F'])
>>> a = ndrange([nat, sex], start=1)
>>> a
nat\sex | M | F
      BE | 1 | 2
      FO | 3 | 4
>>> d = diag(a)
>>> d
nat,sex | BE,M | FO,F
      |    1 |    4
>>> diag(d)
nat\sex | M | F
      BE | 1 | 0
      FO | 0 | 4
>>> a = ndrange(sex, start=1)
>>> a
sex | M | F
    | 1 | 2
>>> diag(a)
sex\sex | M | F
      M | 1 | 0
      F | 0 | 2
```

For completeness

- added `Axis.rename` method which returns a copy of the axis with a different name and deprecate `Axis._rename`
- added `labels_array` as a generalized version of `identity` (which is deprecated)
- implemented `LArray.ipoints[...]` to do point selection using coordinates instead of labels (aka numpy indexing)
- raise an error when trying to do `a[key_with_more_axes_than_a] = value` instead of silently ignoring extra axes.
- allow using a single int for `index_col` in `read_csv` in addition to a list of ints
- implemented `__getitem__` for “x”. You can now write stuff like:

```
>>> a = ndrange((3, 4))
>>> a[x[0][1:]]
{0}\{1}* | 0 | 1 | 2 | 3
          | 4 | 5 | 6 | 7
          | 8 | 9 |10 |11
>>> a[x[1][2:]]
{0}* \{1} | 2 | 3
          | 0 | 2 | 3
          | 1 | 6 | 7
          | 2 |10 |11
>>> a.sum(x[0])
{0}* | 0 | 1 | 2 | 3
      |12 |15 |18 |21
```

- produce normal axes instead of wildcard axes on `LArray.points[...]`. This is (much) slower but more correct/informative.
- changed the way we store axes internally, which has several consequences
 - better overall support for anonymous axes
 - better support for arrays with several axes with the same name
 - small performance improvement
 - the same axis object cannot be added twice in an array (one should use `axis.copy()` if that need arises)
 - changes the way groups with an axis are displayed
- fixed `sum`, `min`, `max` functions on non-LArray arguments
- changed `__repr__` for wildcard axes to not display their labels but their length

```
>>> ndrange(3).axes[0]
Axis(None, 3)
```

- fixed aggregates on several groups “forgetting” the name of groups which had been created using `axis.all()`
- allow `Axis(..., long)` in addition to `int` (Python2 only)
- better docstrings/tests/comments/error messages/thoughts/...

6.1.28 Version 0.11.1

Released on 2016-05-25.

Fixes

- fixed new functions `full`, `full_like` and `create_sequential` not being available when using `from larray import *`

6.1.29 Version 0.11

Released on 2016-05-25.

Viewer

- implemented “Copy to Excel” in context menu (Ctrl+E), to open the selection in a new Excel sheet directly, without the need to use paste. If nothing is selected, copies the whole array.
- when nothing is selected, Ctrl C selects & copies the whole array to the clipboard.
- when nothing is selected, Ctrl V paste at top-left corner
- implemented `view(dict_with_array_values)`

```
>>> view({'a': array1, 'b': array2})
```

- fixed copy (ctrl-C) when viewing a 2D array: it did not include labels from the first axis in that case

Core

- implemented `LArray.growth_rate` to compute the growth along an axis

```
>>> sex = Axis('sex', ['M', 'F'])
>>> year = Axis('year', [2015, 2016, 2017])
>>> a = ndrange([sex, year]).cumsum(x.year)
>>> a
sex\year | 2015 | 2016 | 2017
      M |    0 |    1 |    3
      F |    3 |    7 |   12
>>> a.growth_rate()
sex\year |      2016 |      2017
      M |      inf |      2.0
      F | 1.333333333333 | 0.714285714286
>>> a.growth_rate(d=2)
sex\year | 2017
      M |  inf
      F |  3.0
```

- implemented `LArray.diff` (difference along an axis)

```
>>> sex = Axis('sex', ['M', 'F'])
>>> xtype = Axis('type', ['type1', 'type2', 'type3'])
>>> a = ndrange([sex, xtype]).cumsum(x.type)
>>> a
sex\type | type1 | type2 | type3
```

(continues on next page)

(continued from previous page)

```

      M |    0 |    1 |    3
      F |    3 |    7 |   12
>>> a.diff()
sex\type | type2 | type3
      M |    1 |    2
      F |    4 |    5
>>> a.diff(n=2)
sex\type | type3
      M |    1
      F |    1
>>> a.diff(x.sex)
sex\type | type1 | type2 | type3
      F |    3 |    6 |    9

```

- implemented `round()` (as a nicer alias to `around()` and `round_()`)

```

>>> a = ndrange(5) + 0.5
>>> a
axis0 | 0 | 1 | 2 | 3 | 4
      | 0.5 | 1.5 | 2.5 | 3.5 | 4.5
>>> round(a)
axis0 | 0 | 1 | 2 | 3 | 4
      | 0.0 | 2.0 | 2.0 | 4.0 | 4.0

```

- implemented `Session[['list', 'of', 'str']]` to get a subset of a `Session`

```

>>> s = Session({'a': ndrange(3), 'b': ndrange(4), 'c': ndrange(5)})
>>> s
Session(a, b, c)
>>> s['a', 'c']
Session(a, c)

```

- implemented `LArray.points` to do pointwise indexing instead of the default orthogonal indexing when indexing several dimensions at the same time.

```

>>> a = Axis('a', ['a1', 'a2', 'a3'])
>>> b = Axis('b', ['b1', 'b2', 'b3'])
>>> arr = ndrange((a, b))
>>> arr
a\b | b1 | b2 | b3
a1 | 0 | 1 | 2
a2 | 3 | 4 | 5
>>> arr[['a1', 'a3'], ['b1', 'b2']]
a\b | b1 | b2
a1 | 0 | 1
a3 | 6 | 7
# this selects the points ('a1', 'b1') and ('a3', 'b2')
>>> arr.points[['a1', 'a3'], ['b1', 'b2']]
a,b* | 0 | 1
      | 0 | 7

```

Note that `.ipoints` (to do pointwise indexing with positions instead of labels – aka numpy indexing) is planned but not functional yet.

- made “`arr1.drop_labels() * arr2`” use the labels from `arr2` if any


```

>>> a = Axis('a', ['a1', 'a2'])
>>> b = Axis('b', ['b1', 'b2'])
>>> b2 = Axis('b', ['b2', 'b3'])
>>> arr1 = ndrange([a, b])
>>> arr1
a\b | b1 | b2
a1 | 0 | 1
a2 | 2 | 3
>>> arr1.drop_labels(b)
a\b* | 0 | 1
    a1 | 0 | 1
    a2 | 2 | 3
>>> arr1.drop_labels([a, b])
a\b* | 0 | 1
    0 | 0 | 1
    1 | 2 | 3
>>> arr2 = ndrange([a, b2])
>>> arr2
a\b | b2 | b3
a1 | 0 | 1
a2 | 2 | 3
>>> arr1 * arr2
Traceback (most recent call last):
...
ValueError: incompatible axes:
Axis('b', ['b2', 'b3'])
vs
Axis('b', ['b1', 'b2'])
>>> arr1 * arr2.drop_labels()
a\b | b1 | b2
a1 | 0 | 1
a2 | 4 | 9
# in versions < 0.11, it used to return:
# >>> arr1.drop_labels() * arr2
# a\b* | 0 | 1
#    0 | 0 | 1
#    1 | 2 | 3
>>> arr1.drop_labels() * arr2
a\b | b2 | b3
a1 | 0 | 1
a2 | 4 | 9
>>> arr1.drop_labels('a') * arr2.drop_labels('b')
a\b | b1 | b2
a1 | 0 | 1
a2 | 4 | 9

```

- made `.plot` a property, like in Pandas, so that we can do stuff like:

```

>>> a.plot.bar()
# instead of
>>> a.plot(kind='bar')

```

- made labels from different types not match against each other even if their value is the same. This might break some code but it is both more efficient and more convenient in some cases, so let us see how it goes:

```

>>> a = ndrange(4)
>>> a

```

(continues on next page)

(continued from previous page)

```

axis0 | 0 | 1 | 2 | 3
      | 0 | 1 | 2 | 3
>>> a[1]
1
>>> # This used to "work" (and return 1)
>>> a[True]
...
ValueError: True is not a valid label for any axis

```

```

>>> a[1.0]
...
ValueError: 1.0 is not a valid label for any axis

```

- implemented `read_csv(dialect='liam2')` to read .csv files formatted like in LIAM2 (with the axes names on a separate line than the last axis labels)
- implemented `Session[boolean LArray]`

```

>>> a = ndrange(3)
>>> b = ndrange(4)
>>> s1 = Session({'a': a, 'b': b})
>>> s2 = Session({'a': a + 1, 'b': b})
>>> s1 == s2
name |      a |      b
      | False | True
>>> s1[s1 == s2]
Session(b)
>>> s1[s1 != s2]
Session(a)

```

- implemented experimental support for creating an array sequentially. Comments on the name of the function and syntax (especially compared to `ndrange`) would be appreciated.

```

>>> year = Axis('year', range(2016, 2020))
>>> sex = Axis('sex', ['M', 'F'])
>>> create_sequential(year)
year | 2016 | 2017 | 2018 | 2019
      | 0 | 1 | 2 | 3
>>> create_sequential(year, 1.0, 0.1)
year | 2016 | 2017 | 2018 | 2019
      | 1.0 | 1.1 | 1.2 | 1.3
>>> create_sequential(year, 1.0, mult=1.1)
year | 2016 | 2017 | 2018 | 2019
      | 1.0 | 1.1 | 1.21 | 1.331
>>> inc = LArray([1, 2], [sex])
>>> inc
sex | M | F
      | 1 | 2
>>> create_sequential(year, 1.0, inc)
sex\year | 2016 | 2017 | 2018 | 2019
          M | 1.0 | 2.0 | 3.0 | 4.0
          F | 1.0 | 3.0 | 5.0 | 7.0
>>> mult = LArray([2, 3], [sex])
>>> mult
sex | M | F
      | 2 | 3

```

(continues on next page)

(continued from previous page)

```

>>> create_sequential(year, 1.0, mult=mult)
sex\year | 2016 | 2017 | 2018 | 2019
      M | 1.0 | 2.0 | 4.0 | 8.0
      F | 1.0 | 3.0 | 9.0 | 27.0
>>> initial = LArray([3, 4], [sex])
>>> initial
sex | M | F
    | 3 | 4
>>> create_sequential(year, initial, inc, mult)
sex\year | 2016 | 2017 | 2018 | 2019
      M | 3 | 7 | 15 | 31
      F | 4 | 14 | 44 | 134
>>> def modify(prev_value):
...     return prev_value / 2
>>> create_sequential(year, 8, func=modify)
year | 2016 | 2017 | 2018 | 2019
     | 8 | 4 | 2 | 1
>>> create_sequential(3)
axis0* | 0 | 1 | 2
       | 0 | 1 | 2
>>> create_sequential(x.year, axes=(sex, year))
sex\year | 2016 | 2017 | 2018 | 2019
      M | 0 | 1 | 2 | 3
      F | 0 | 1 | 2 | 3

```

- implemented full and full_like to create arrays initialize to something else than zeros or ones

```

>>> nat = Axis('nat', ['BE', 'FO'])
>>> sex = Axis('sex', ['M', 'F'])
>>> full([nat, sex], 42.0)
nat\sex | M | F
      BE | 42.0 | 42.0
      FO | 42.0 | 42.0
>>> initial_value = ndrange([sex])
>>> initial_value
sex | M | F
    | 0 | 1
>>> full([nat, sex], initial_value)
nat\sex | M | F
      BE | 0 | 1
      FO | 0 | 1

```

- performance improvements when using label keys: a[key] is faster, especially if key is large

Fixes

- `to_excel(filepath)` only closes the file if it was not open before
- removed code which forced labels from .csv files to be strings (as it caused problems in many cases, e.g. ages in LIAM2 files)

Misc. stuff for completeness

- made LGroups usable in Python's builtin `range()` and convertible to int and float
- implemented `AxisCollection.union` (equivalent to `AxisCollection | Axis`)
- fixed boolean array keys (boolean filter) in combination with scalar keys (for other dimensions)
- fixed support for older numpy
- fixed `LArray.shift(n=0)`
- still more work on making arrays with anonymous axes usable (not there yet)
- added more tests
- better docstrings/error messages...
- misc. code cleanup/simplification/improved comments

6.1.30 Version 0.10.1

Released on 2016-03-25.

New features

- A single change in this release: a much more powerful `to_excel` function which (by default) use Excel itself to write files. Additional functionality include:
 - write in an existing file without overwriting existing data/sheet/...
 - write at a precise position
 - view an array in a live Excel instance (a new OR an existing workbook)

See `to_excel()` documentation for details.

6.1.31 Version 0.10

Released on 2016-03-22.

Core

- implemented dropna argument for to_csv, to_frame and to_series to avoid writing lines with either 'all' or 'any' NA values.
- implemented read_sas. Needs pandas >= 0.18 (though it seems still buggy on some files).
- implemented experimental support for __getattr__ and __setattr__ on LArray. One can use arr.H instead of arr['M']. It only works for single string labels though (not for slices or list of labels nor integer labels). Not sure it is a good idea :).
- **implemented Session +/-** Eg. sess1 - sess2 will compute the difference on each array present in either session. If an array is present in one session and not in the other, it is replaced by "NaN".
- added .nbytes property to LArray objects (to know how many bytes of memory the array uses)
- made sort_axis accept a tuple of axes
- raises an error on a.i[tuple_with_len_greater_than_array_ndim]
- slightly better support for axes with no name (no, still no complete support yet ;-))
- improved AxisCollection: implemented __delitem__(slice), __setitem__(list), __setitem__(slice)
- fixed exception on AxisCollection.index(invalid_index)
- better docstrings for a few functions
- misc code cleanups, refactoring & improved tests

Editor

- added .dirty property on ArrayEditorWidget
- fixed viewing arrays with "inf" (infinite)
- fixed a few edge cases for the ndigit detection code
- fixed colors in some cases in edit()
- made copy-paste of large regions faster in some cases

6.1.32 Version 0.9.2

Released on 2016-03-02.

Core

- much better support for unnamed axes overall. Still a long way to go for full support, but it's getting there...

Editor

- fixed edit() for arrays with the same labels on several axes

6.1.33 Version 0.9.1

Released on 2016-03-01.

Core

- better .info for arrays with groups in axes

```
>>> # example using groups without a name
>>> reg = la.sum((fla, wal, bru, belgium))
>>> reg.info
4 x 15
geo [4]: ['A11' ... 'A73'] ['A25' ... 'A93'] 'A21' ['A11' ... 'A21']
lipro [15]: 'P01' 'P02' 'P03' ... 'P13' 'P14' 'P15'
```

```
>>> # example using groups with a name
>>> fla = geo.group(fla_str, name='Flanders')
>>> wal = geo.group(wal_str, name='Wallonia')
>>> bru = geo.group(bru_str, name='Brussels')
>>> reg = la.sum((fla, wal, bru))
>>> reg.info
3 x 15
geo [3]: 'Flanders' (['A11' ... 'A73']) 'Wallonia' (['A25' ... 'A93']) 'Brussels
↪' ('A21')
lipro [15]: 'P01' 'P02' 'P03' ... 'P13' 'P14' 'P15'
```

Editor

- fixed edit() with non-string labels in axes
- fixed edit() with filters in some more cases
- fixed ArrayEditorWidget.reject_changes and accept_changes to update the model & view accordingly (in case the widget is kept open)
- avoid (harmless) error messages in some cases

6.1.34 Version 0.9

Released on 2016-02-25.

A minor but backward incompatible version (hence the bump in version number)!

Core

- fixed `int_array.mean()` to return floats instead of int (regression in 0.8)
- `larray_equal` returns False when either value is not an LArray, instead of raising an exception

Session

- changed `Session == Session` to return an array of booleans instead of a single boolean, so that we know which array(s) differ. Code like `session1 == session2`, should be changed to `all(session1 == session2)`.
- implemented `Session != Session`
- implemented `Session.get(k, default)` (returns default if k does not exist in Session)
- implemented `len()` for Session objects to know how many objects are in the Session

Viewer

- fixed `view()` (regression in 0.8.1)
- fixed `edit()` to actually apply changes on “OK”/accept_changes even when no filter change occurred after the last edit.

6.1.35 Version 0.8.1

Released on 2016-02-24.

Viewer

- implemented min/maxvalue arguments for `edit()`
- do not close the window when pressing Enter
- allow to start editing cells by pressing Enter
- fixed copy of changed cells (copy the changed value)
- fixed pasted values to not be accepted directly (they go to “changes” like for manual edits)
- fixed color updates on paste
- disabled experimental tooltips on headers
- better error message when entering invalid values

Core

- implemented indexing by position on several dimensions at once (like numpy)

```
>>> # takes the first item in the first and third dimensions, leave the second
↳dimension intact
>>> arr.i[0, :, 0]
<some result>
>>> # sets all the cells corresponding to the first item in the first dimension
↳and the second item in the fourth
```

(continues on next page)

(continued from previous page)

```
>>> # dimension
>>> arr.i[0, :, :, 1] = 42
```

- added optional ‘readonly’ argument to `expand()` to produce a readonly view (much faster since no copying is done)

6.1.36 Version 0.8

Released on 2016-02-16.

Core

- implemented `skipna` argument for most aggregate functions. defaults to `True`.
- implemented `LArray.sort_values(key)`
- implemented percentile and median
- added `isnan` and `isinf` toplevel functions
- made `axis` argument optional for `argsort` & `posargsort` on 1D arrays
- fixed `a[key] = value` when `key` corresponds to a single cell of the array
- fixed `keepaxes` argument for aggregate functions
- fixed `a[int_array]` (when the axis needs to be guessed)
- fixed `empty_like`
- fixed aggregates on several axes given as integers e.g. `arr.sum(axis=(0, 2))`
- fixed “kind” argument in `posargsort`

Viewer

- added `title` argument to `edit()` (set automatically if not provided, like for `view()`)
- fixed `edit()` on filtered arrays
- fixed `view(expression)`. anything which was not stored in a variable was broken in 0.7.1
- reset background color when setting values if necessary (still buggy in some cases, but much less so ;-))
- background color for headers is always on
- `view()` => array cells are not editable, instead of being editable and ignoring entered values
- fixed `compare()` colors when arrays are entirely equal
- fixed error message for `compare()` when `PyQt` is not available

Misc

- bump numpy requirement to 1.10, implicitly dropping support for python 3.3
- renamed view module to editor to not collide with view function
- improved/added a few tests

6.1.37 Version 0.7.1

Released on 2016-01-29.

Viewer

- implemented paste (ctrl-V)
- implemented experimental array comparator:

```
>>> compare(array1, array2)
```

Known limitation: the arrays must have exactly the same axes and the background color is buggy when using filters

- when no title is specified in view(), it is determined automatically by inspecting the local variables of the function where view() is called and using the names of the ones matching the object passed. If several matches, up to 3 are displayed.
- added axes names to copy (ctrl-C)
- fixed copy (ctrl-C) of 0d array

Input/Output

- added 'dialect' argument to to_csv. For example, dialect='classic' does not include the last (horizontal) axis name.
- fixed loading .csv files without (ie 'classic' .csv files), though one needs to specify nb_index in that case if ndim > 2
- strip spaces around axes names so that you can use "axis0<space><space>axis1" instead of "axis0axis1" in .csv files
- fixed 1d arrays I/O
- more precise parsing of input headers: 1 and 0 come out as int, not bool

Misc

- nicer error message when using an invalid axes names
- changed LArray .df property to a to_frame() method so that we can pass options to it

6.1.38 Version 0.7

Released on 2016-01-26.

Viewer

- implemented `view()` on `Session` objects
- added axes length in window title and add axes info even if title is provided manually (concatenate both)
- `ndecimal`s are recomputed when toggling the scientific checkbox
- allow viewing (some) non-ndarray stuff (e.g. python lists)
- refactored viewer code so that the filter drop downs can be reused too
- Known regression: the viewer is slow on large arrays (this will be fixed in a later release, obviously)

Session

- implemented `local_arrays()` to return all `LArray` in `locals()` as a `Session`
- implemented `Session.__getitem__(int_position)`
- implement `Session(filename)` to directly load all arrays from a file. Equivalent to:

```
>>> s = Session()
>>> s.load(filename)
```

- implemented `Session.__eq__`, so that you can compare two sessions and see if all arrays are equal. Suppose you want to refactor your code and make sure you get the same results.

```
>>> # put results in a Session
>>> res = Session({'array1': array1, 'array2': array2})
>>> # before refactoring
>>> res.dump('results.h5')
>>> # after refactoring
>>> assert Session('results.h5') == res
```

- you can load all sheets/arrays of a file (if you do not specify which ones you want, it takes all)
- loading several sheets from an excel file is now MUCH faster because the same file is kept open (apparently `xlrd` parses the whole file each time we open it).
- you can specify a subset of arrays to dump
- implemented rudimentary session I/O for `.csv` files, usage is a bit different from `.h5` & excel files

```
>>> # need to specify format manually
>>> s.dump('directory_name', fmt='csv')
>>> # need to specify format manually
>>> s = Session()
>>> s.load('directory_name', fmt='csv')
```

- pass `*args` and `**kwargs` to lower level functions in `Session.load`
- fail when trying to read an inexistant H5 file through `Session`, instead of creating it

Other new features

- added start argument in ndrange to specify starting value
- implemented Axis._rename. Not sure it's a good idea though...
- implemented identity function which takes an Axis and returns an LArray with the axis labels as values
- implemented size property on AxisCollection
- allow a single int in AxisCollection.without

Fixes

- fixed broadcast_with when other_axes contains 0-len axes
- fixed a[bool_array] = value when the first axis of a is not in bool_array
- fixed view() on arrays with unnamed axes
- fixed view() on arrays of Python objects
- various other small bugs fixed

6.1.39 Version 0.6.1

Released on 2016-01-13.

New features

- added dtype argument to all array creation functions to override default data type
- aggregates can take an explicit “axis” keyword argument which can be used to target an axis by index

```
>>> arr.sum(axis=0)
```

- implemented LGroup.__getitem__ & LGroup.__iter__, so that for list-based groups (ie not slices) you can write:

```
>>> for v in my_group:
...     # some code
```

or

```
>>> my_group[0]
```

Miscellaneous improvements

- renamed LabelGroup to LGroup and PositionalKey to PGroup. We might want to rename the later to IGroup (to be consistent with axis.i[...]).
- slightly better support for axes without name
- better docstrings for a few functions
- misc cleanup

Fixes

- fixed XXX_like(a) functions to use the same dtype than a instead of always float
- fixed to_XXX with 1d arrays (e.g. to_clipboard())
- fixed all() and any() toplevel functions without argument
- fixed LArray without axes in some cases
- fixed array creation functions with only shapes on python2

6.1.40 Version 0.6

Released on 2016-01-12.

New features

- a[bool_array_key] broadcasts missing/differently ordered dimensions and returns an LArray with combined axes
- a[bool_array_key] = value broadcasts missing/differently ordered dimensions on both key and value
- **implemented argmin, argmax, argsort, posargmin, posargmax, posargsort.** they do indirect operation along an axis. E.g. argmin gives the label of the minimum value, argsort gives the labels which would sort the array along that dimension. posargXXX gives the position/indexes instead of the labels.
- implemented Axis.__iter__ so that one can write:

```
>>> for label in an_array.axes.an_axis:
...     <some code>
```

instead of

```
>>> for label in an_array.axes.an_axis.labels:
...     <some code>
```

- implemented the .info property on AxisCollection
- implement all/any top level functions, so that you can use them in with_total.

Miscellaneous improvements

- renamed ValueGroup to LabelGroup. We might want to rename it to LGroup to be consistent with LArray?
- allow a single int as argument to LArray creation functions (ndrange et al.)
e.g. *ndrange(10)* is now allowed instead of *ndrange([10])*
- use display_name in .info (ie add * next to wildcard axes in .info).
- allow specifying a custom window title in view()
- viewer displays booleans as True/False instead of 1/0
- slightly better support for axes with no name (None). There is still a long way to go for full support though.
- improved a few docstrings
- nicer errors when tests results are different from expected

- removed debug prints from viewer
- misc cleanups

Fixes

- fixed view() on all-negative arrays
- fixed view() on string arrays

6.1.41 Version 0.5

Released on 2015-12-15.

New features

- experimental support for indexing an LArray by another (integer) LArray

```
>>> array[other_array]
```

- experimental support for LArray.drop_labels and the concept of wildcard axes
- added LArray.display_name and AxisCollection.display_names which add '*' next to wildcard axes
- implemented where(cond, array1, array2)
- implemented LArray.__iter__ so that this works:

```
>>> for value in array:
...     <some code>
```

- implement keepaxes=label or keepaxes=True for aggregate functions on full axes
array.sum(x.age, keepaxes='total')
- AxisCollection.replace can replace several axes in one call
- implemented .expand(out=) to expand into an existing array

Miscellaneous improvements

- removed Axis.sorted()
- removed LArray.axes_names & axes_labels. One should use .axes.names & .axes.labels instead.
- raise an error when trying to convert an array with more than one value to a Boolean. For example, this will fail:

```
>>> arr = ndrange([sex])
>>> if arr:
...     <some code>
```

- convert value to self.dtype in append/prepend
- faster .extend, .append, .prepend and .expand
- some code cleanup, better tests, ...

Fixes

- fixed `.extend` when other has longer axes than self

6.1.42 Version 0.4

Released on 2015-12-09.

New features

- implemented `LArray.expand` to add dimensions
- implemented `prepend`
- implemented `sort_axis`
- allow creating 0d (scalar) LArrays

Miscellaneous improvements

- made `extend` expand its arguments
- made `.append` expand its value before appending
- changed `read_*` to not sort data by default
- more minor stuff :)

Fixes

- fixed loading 1d arrays

6.1.43 Version 0.3

Released on 2015-11-26.

New features

- implemented `LArray.with_total()`: appends axes or group aggregates to the array.

Without argument, it adds totals on all axes. It has optional keyword only arguments:

- *label*: specify the label (“total” by default)
- *op*: specify the aggregate function (sum by default, all other aggregates should work too)

With multiple arguments, it adds totals sequentially. There are some tricky cases. For example when, for the same axis, you add group aggregates and axis aggregates:

```
>>> # works but "wrong" for x.geo (double what is expected because the total also
>>> # includes fla wal & bru)
>>> la.with_total(x.sex, (fla, wal, bru), x.geo, x.lipro)
```

```
>>> # correct total but the order is not very nice
>>> la.with_total(x.sex, x.geo, (fla, wal, bru), x.lipro)
```

```
>>> # the correct way to do it, but it is probably not entirely obvious
>>> la.with_total(x.sex, (fla, wal, bru, x.geo.all()), x.lipro)
```

```
>>> # we probably want to display a warning (or even an error?) in that case.
>>> # If the user really wants that behavior, he can split the operation:
>>> # .with_total((fla, wal, bru)).with_total(x.geo)
```

- implemented group aggregates without using keyword arguments. As a consequence of this, one can no longer use axis numbers in aggregates. Eg. `a.sum(0)` does not sum on the first axis anymore (but you can do `a.sum(a.axes[0])` if needed)
- implemented `LArray.percent`: equivalent to `ratio * 100`
- implemented `Session.filter` -> returns a new `Session` with only objects matching the filter
- implemented `Session.dump` -> dumps all `LArray` in the `Session` to a file
- implemented `Session.load` -> load several `LArrays` from a file to a `Session`

6.1.44 Version 0.2.6

Released on 2015-11-24.

Fixes

- fixed `LArray.cumsum` and `cumprod`.
- fixed all doctests just enough so that they run.

6.1.45 Version 0.2.5

Released on 2015-10-29.

Miscellaneous improvements

- many methods got (improved) docstrings (Thanks to Johan).

Fixes

- fixed mixing keys without axis (e.g. `arr[10:15]`) with key with axes (e.g. `arr[x.age[10:15]]`).

6.1.46 Version 0.2.4

Released on 2015-10-27.

New features

- includes an experimental (slightly inefficient) version of guess axis, so that one can write:

```
>>> arr[10:20]
```

instead of

```
>>> arr[age[10:20]]
```

6.1.47 Version 0.2.3

Released on 2015-10-19.

New features

- positional slicing via “x.” syntax (x.axis.i[:5])

Fixes

- view(array) is usable when doing *from larray import **
- fixed a nasty bug for doing “group” aggregates when there is only one dimension

6.1.48 Version 0.2.2

Released on 2015-10-15.

New features

- implement AxisCollection.replace(old_axis, new_axis)
- implement positional indexing

Miscellaneous improvements

- more powerful AxisCollection.pop added support .pop(name) or .pop(Axis object)
- LArray.set_labels returns a new LArray by default use inplace=True to get previous behavior
- include ndrange and __version__ in __all__

Fixes

- fixed shift with $n \leq 0$

6.1.49 Version 0.2.1

Released on 2015-10-14.

New features

- implemented `LArray.shift(axis, n=1)`

Miscellaneous improvements

- change `set_labels` API (`axis`, `new_labels`)
- transform `Axis.labels` into a property so that `_mapping` is kept in sync

Fixes

- hopefully fix build

6.1.50 Version 0.2

Released on 2015-10-13.

New features

- added `to_clipboard`.
- added embryonic documentation.
- added `sort_columns` and `na` arguments to `read_hdf`.
- added `sort_rows`, `sort_columns` and `na` arguments to `read_excel`.
- added `setup.py` to install the module.

Miscellaneous improvements

- IO functions (`to_*/read_*`) now support unnamed axes. The set of supported operations is very limited with such arrays though.
- `to_excel` `sheet_name` defaults to “Sheet1” like in Pandas.
- reorganised files.
- automated somewhat releases (added a rudimentary release script).

Fixes

- column titles are no longer converted to lowercase.

6.1.51 Version 0.1

Released on 2014-10-22.

6.2 How to contribute

6.2.1 Before Starting

Where to find the code

The code is hosted on [GitHub](#).

Tools

To contribute you will need to sign up for a [free GitHub account](#).

We use [Git](#) for version control to allow many people to work together on the project.

The documentation is written partly using reStructuredText and partly using Jupyter notebooks (for the tutorial). It is built to various formats using [Sphinx](#) and [nbsphinx](#).

The unit tests are written using the [pytest library](#). The compliance with the PEP8 conventions is tested using the extension [pytest-pep8](#).

Many editors and IDE exist to edit Python code and provide integration with version control tools (like git). A good IDE, such as PyCharm, can make many of the steps below much more efficient.

Licensing

LArray is licensed under the GPLv3. Before starting to work on any issue, make sure you accept and are allowed to have your contributions released under that license.

6.2.2 Creating a development environment

Getting started with Git

[GitHub](#) has [instructions](#) for installing and configuring git.

Getting the code (for the first time)

You will need your own fork to work on the code. Go to the [larray project page](#) and hit the Fork button.

You will want to clone your fork to your machine. To do it manually, follow these steps:

```
git clone https://github.com/your-user-name/larray.git
cd larray
git remote add upstream https://github.com/larray-project/larray.git
```

This creates the directory *larray* and connects your repository to the upstream (main project) *larray* repository. You can see the remote repositories:

```
git remote -v
```

If you added the upstream repository as described above you will see something like:

```
origin  git@github.com:yourname/larray.git (fetch)
origin  git@github.com:yourname/larray.git (push)
upstream      git://github.com/larray-project/larray.git (fetch)
upstream      git://github.com/larray-project/larray.git (push)
```

Creating a Python Environment

Before starting any development, you will need a working Python installation. It is recommended (but not required) to create an isolated larray development environment. One of the easiest way to do it is via *Anaconda* or *Miniconda*:

- Install either [Anaconda](#) or [miniconda](#) as *suggest earlier*
- Make sure your conda is up to date (`conda update conda`)
- Make sure that you have *cloned the repository*
- `cd` to the *larray* source directory

We'll now kick off a two-step process:

1. Install the build dependencies

```
# add 'conda-forge' channel (required to install some dependencies)
conda config --add channels conda-forge

# Create and activate the build environment
conda create -n larray_dev numpy pandas pytables pyqt qtpy matplotlib xlrd openpyxl
↳xlswriter pytest pytest-pep8
conda activate larray_dev
```

This will create the new environment, and not touch any of your existing environments, nor any existing Python installation.

To view your environments:

```
conda info -e
```

To return to your root environment:

```
conda deactivate
```

See the full conda docs [here](#).

2. Build and install larray

Install larray using the following command:

```
python setup.py develop
```

This creates some kind of symbolic link between your python installation “modules” directory and your repository, so that any change in your local copy is automatically usable by other modules.

At this point you should be able to import larray from your local version:

```
$ python # start an interpreter
>>> import larray
>>> larray.__version__
'0.29-dev'
```

6.2.3 Starting to contribute

With your local version of larray, you are now ready to contribute to the project. To make a contribution, please follow the steps described below.

Step 1: Create a new branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git checkout -b issue123
```

This changes your working directory to the issue123 branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to the project. You can have many different branches and switch between them using the `git checkout` command.

To update this branch, you need to retrieve the changes from the master branch:

```
git fetch upstream
git rebase upstream/master
```

This will replay your commits on top of the latest larray git master. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to `stash` them prior to updating. This will effectively store your changes and they can be reapplied after updating.

Step 2: Write your code

When writing your code, please follow the [PEP8](#) code conventions. Among others, this means:

- 120 characters lines
- 4 spaces indentation
- lowercase (with underscores if needed) variables, functions, methods and modules names
- CamelCase classes names
- all uppercase constants names
- whitespace around binary operators
- no whitespace before a comma, semicolon, colon or opening parenthesis

- whitespace after commas

This summary should not prevent you from reading the PEP!

LArray is currently compatible with both Python 2 and 3. So make sure your code is compatible with both versions.

Step 3: Document your code

We use Numpy conventions for docstrings. Here is a template:

```
def funcname(arg1, arg2=default2, arg3=default3):
    """Summary line.

    Extended description of function.

    .. versionadded:: 0.2.0

    Parameters
    -----
    arg1 : type1
        Description of arg1.
    arg2 : {value1, value2, value3}, optional
        Description of arg2.

        * value1 -- description of value1 (default2)
        * value2 -- description of value2
        * value3 -- description of value3
    arg3 : type3 or type3bis, optional
        Description of arg3. Default is default3.

    .. versionadded:: 0.3.0

    Returns
    -----
    type
        Description of return value.

    Notes
    -----
    Some interesting facts about this function.

    See Also
    -----
    LArray.otherfunc : How other function or method is related.

    Examples
    -----
    >>> funcname(arg)
    result
    """
```

For example:

```
def check_number_string(number, string="1"):
    """Compares the string representation of a number to a string.

    Parameters
    -----
```

(continues on next page)

(continued from previous page)

```

number : int
    The number to test.
string : str, optional
    The string to test against. Default is "1".

Returns
-----
bool
    Whether the string representation of the number is equal to the string.

Examples
-----
>>> check_number_string(42, "42")
True
>>> check_number_string(25, "2")
False
>>> check_number_string(1)
True
"""
return str(number) == string

```

Step 4: Test your code

Our unit tests are written using the [pytest library](#) and our tests modules are located in `/larray/tests/`. We also use its extension `pytest-pep8` to check if the code is PEP8 compliant. The pytest library is able to automatically detect and run unit tests as long as you respect some conventions:

- pytest will search for `test_*.py` or `*_test.py` files.
- From those files, collect test items:
 - `test_` prefixed test functions or methods outside of class.
 - `test_` prefixed test functions or methods inside Test prefixed test classes (without an `__init__` method).

For more details, please read the section [Conventions for Python test discovery](#) from the [pytest documentation](#).

Here is an example of a unit test function using pytest:

```

from larray.core.axis import _to_key

def test_key_string_split():
    assert _to_key('M,F') == ['M', 'F']
    assert _to_key('M,') == ['M']

```

To run unit tests for a given test module:

```
> pytest larray/tests/test_array.py
```

We also use doctests for some tests. Doctests is specially-formatted code within the docstring of a function which embeds the result of calling said function with a particular set of arguments. This can be used both as documentation and testing. We only use doctests for the cases where the test is simple enough to fit on one line and it can help understand what the function does. For example:

```

def slice_to_str(key):
    """Converts a slice to a string

```

(continues on next page)

(continued from previous page)

```
>>> slice_to_str(slice(None))
': '
"""
# some clever code here
return ': '
```

To run doc tests:

```
> pytest larray/core/array.py
```

To run all the tests, simply go to root directory and type:

```
> pytest
```

pytest will automatically detect all existing unit tests and doctests and run them all.

Step 5: Add a change log

Changes should be reflected in the release notes located in `doc/source/changes/version_<next_release_version>.inc`. This file contains an ongoing change log for the next release. Add an entry to this file to document your fix, enhancement or (unavoidable) breaking change. If you hesitate in which section to add your change log, feel free to ask. Make sure to include the GitHub issue number when adding your entry (using closes `:issue:`123`` where 123 is the number associated with the fixed issue).

Step 6: Commit your changes

When all the above is done, commit your changes. Make sure that one of your commit messages starts with `fix #123` : (where 123 is the issue number) before starting any pull request (see [this github page](#) for more details).

Step 7: Push your changes

When you want your changes to appear publicly on the web page of your fork on GitHub, push your forked feature branch's commits:

```
git push origin issue123
```

Here `origin` is the default name given to your remote repository on GitHub.

Step 8: Start a pull request

You are ready to request your changes to be included in the master branch (so that they will be available in the next release). To submit a pull request:

1. Navigate to your repository on GitHub
2. Click on the Pull Request button
3. You can then click on Commits and Files Changed to make sure everything looks okay one last time
4. Write a description of your changes in the Preview Discussion tab
5. If this is your first pull request, please state explicitly that you accept and are allowed to have your contribution (and any future contribution) licensed under the GPL license (See section [Licensing](#) above).

6. Click Send Pull Request.

This request then goes to the repository maintainers, and they will review the code. Your modifications will also be automatically tested by running the *larray* test suite on [Travis-CI](#) continuous integration service. A pull request will only be considered for merging when you have an all ‘green’ build. If any tests are failing, then you will get a red ‘X’, where you can click through to see the individual failed tests.

If you need to make more changes to fix test failures or to take our comments into account, you can make them in your branch, add them to a new commit and push them to GitHub using:

```
git push origin issue123
```

This will automatically update your pull request with the latest code and trigger the automated tests again.

Warning: Please do not rebase your local branch during the review process.

6.2.4 Documentation

The documentation is written using reStructuredText and built to various formats using [Sphinx](#). See the [reStructured-Text Primer](#) for a first introduction of the syntax.

Installing Requirements

Basic requirements (to generate an .html version of the documentation) can be installed using:

```
> conda install sphinx numpypdoc nbsphinx
```

To build the .pdf version, you need a LaTeX processor. We use [MiKTeX](#).

To build the .chm version, you need [HTML Help Workshop](#).

Generating the documentation

Open a command prompt and go to the documentation directory:

```
> cd doc
```

If you just want to check that there is no syntax error in the documentation and that it formats properly, it is usually enough to only generate the .html version, by using:

```
> make html
```

Open the result in your favourite web browser. It is located in:

```
build/html/index.html
```

If you want to also generate the .pdf and .chm (and you have the extra requirements to generate those), you could use:

```
> buildall
```


BIBLIOGRAPHY

- [1] Wikipedia, “Convolution”, <https://en.wikipedia.org/wiki/Convolution>
- [1] C. W. Clenshaw, “Chebyshev series for mathematical functions”, in *National Physical Laboratory Mathematical Tables*, vol. 5, London: Her Majesty’s Stationery Office, 1962.
- [2] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 379. http://www.math.sfu.ca/~cbm/aands/page_379.htm
- [3] <http://kobesearch.cpan.org/htdocs/Math-Cephes/Math/Cephes.html>
- [1] Weisstein, Eric W. “Sinc Function.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/SincFunction.html>
- [2] Wikipedia, “Sinc function”, https://en.wikipedia.org/wiki/Sinc_function
- [1] Wikipedia, “Exponential function”, https://en.wikipedia.org/wiki/Exponential_function
- [2] M. Abramowitz and I. A. Stegun, “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables,” Dover, 1964, p. 69, http://www.math.sfu.ca/~cbm/aands/page_69.htm
- [1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. <http://www.math.sfu.ca/~cbm/aands/>
- [2] Wikipedia, “Logarithm”. <https://en.wikipedia.org/wiki/Logarithm>
- [1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. <http://www.math.sfu.ca/~cbm/aands/>
- [2] Wikipedia, “Logarithm”. <https://en.wikipedia.org/wiki/Logarithm>
- [1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. <http://www.math.sfu.ca/~cbm/aands/>
- [2] Wikipedia, “Logarithm”. <https://en.wikipedia.org/wiki/Logarithm>
- [1] ISO/IEC standard 9899:1999, “Programming language C.”
- [1] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*. New York, NY: Dover, 1972, pg. 83. <http://www.math.sfu.ca/~cbm/aands/>
- [2] Wikipedia, “Hyperbolic function”, https://en.wikipedia.org/wiki/Hyperbolic_function
- [1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. <http://www.math.sfu.ca/~cbm/aands/>
- [2] Wikipedia, “Inverse hyperbolic function”, <https://en.wikipedia.org/wiki/Arcsinh>
- [1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. <http://www.math.sfu.ca/~cbm/aands/>

- [2] Wikipedia, “Inverse hyperbolic function”, <https://en.wikipedia.org/wiki/Arccosh>
- [1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. <http://www.math.sfu.ca/~cbm/aands/>
- [2] Wikipedia, “Inverse hyperbolic function”, <https://en.wikipedia.org/wiki/Arctanh>
- [1] Wikipedia, “Normal distribution”, http://en.wikipedia.org/wiki/Normal_distribution
- [2] P. R. Peebles Jr., “Central Limit Theorem” in “Probability, Random Variables and Random Signal Principles”, 4th ed., 2001, pp. 51, 51, 125.

Symbols

[__getitem__\(\)](#) (*larray.Axis method*), 141
[__init__\(\)](#) (*larray.Array method*), 194
[__init__\(\)](#) (*larray.Axis method*), 136
[__init__\(\)](#) (*larray.AxisCollection method*), 171
[__init__\(\)](#) (*larray.ExcelReport method*), 401
[__init__\(\)](#) (*larray.IGroup method*), 151
[__init__\(\)](#) (*larray.LGroup method*), 161
[__init__\(\)](#) (*larray.LSet method*), 170
[__init__\(\)](#) (*larray.Metadata method*), 381
[__init__\(\)](#) (*larray.ReportSheet method*), 405
[__init__\(\)](#) (*larray.Session method*), 428
[__init__\(\)](#) (*larray.Workbook method*), 397
[__init__\(\)](#) (*larray.set_options method*), 415

A

[absolute\(\)](#) (*in module larray*), 331
[add\(\)](#) (*larray.Session method*), 438
[add_graph\(\)](#) (*larray.ReportSheet method*), 408
[add_graphs\(\)](#) (*larray.ReportSheet method*), 409
[add_title\(\)](#) (*larray.ReportSheet method*), 407
[align\(\)](#) (*larray.Array method*), 281
[align\(\)](#) (*larray.Axis method*), 147
[align\(\)](#) (*larray.AxisCollection method*), 189
[all\(\)](#) (*larray.Array method*), 290
[all_by\(\)](#) (*larray.Array method*), 292
[angle\(\)](#) (*in module larray*), 373
[any\(\)](#) (*larray.Array method*), 294
[any_by\(\)](#) (*larray.Array method*), 296
[app\(\)](#) (*larray.Workbook method*), 399
[append\(\)](#) (*larray.Array method*), 278
[append\(\)](#) (*larray.AxisCollection method*), 180
[apply\(\)](#) (*larray.Array method*), 217
[apply\(\)](#) (*larray.Axis method*), 145
[apply\(\)](#) (*larray.Session method*), 441
[apply_map\(\)](#) (*larray.Array method*), 219
[arccos\(\)](#) (*in module larray*), 359
[arccosh\(\)](#) (*in module larray*), 371
[arcsin\(\)](#) (*in module larray*), 358
[arcsinh\(\)](#) (*in module larray*), 370
[arctan\(\)](#) (*in module larray*), 361
[arctan2\(\)](#) (*in module larray*), 363

[arctanh\(\)](#) (*in module larray*), 372
[Array](#) (*class in larray*), 192
[arrays\(\)](#) (*in module larray*), 429
[asarray\(\)](#) (*in module larray*), 410
[astype\(\)](#) (*larray.Array method*), 207
[Axis](#) (*class in larray*), 135
[axis_id\(\)](#) (*larray.AxisCollection method*), 176
[AxisCollection](#) (*class in larray*), 171

B

[broadcast_with\(\)](#) (*larray.Array method*), 281
[by\(\)](#) (*larray.Axis method*), 142
[by\(\)](#) (*larray.IGroup method*), 153
[by\(\)](#) (*larray.LGroup method*), 163

C

[ceil\(\)](#) (*in module larray*), 341
[check_compatible\(\)](#) (*larray.AxisCollection method*), 192
[choice\(\)](#) (*in module larray.random*), 459
[clip\(\)](#) (*larray.Array method*), 313
[close\(\)](#) (*larray.Workbook method*), 398
[combine_axes\(\)](#) (*larray.Array method*), 224
[combine_axes\(\)](#) (*larray.AxisCollection method*), 187
[compact\(\)](#) (*larray.Array method*), 272
[compact\(\)](#) (*larray.Session method*), 444
[compare\(\)](#) (*in module larray*), 452
[conj\(\)](#) (*in module larray*), 375
[containing\(\)](#) (*larray.Axis method*), 139
[containing\(\)](#) (*larray.IGroup method*), 157
[containing\(\)](#) (*larray.LGroup method*), 166
[convolve\(\)](#) (*in module larray*), 329
[copy\(\)](#) (*larray.Array method*), 207
[copy\(\)](#) (*larray.Axis method*), 138
[copy\(\)](#) (*larray.AxisCollection method*), 174
[copy\(\)](#) (*larray.Session method*), 435
[copysign\(\)](#) (*in module larray*), 377
[cos\(\)](#) (*in module larray*), 356
[cosh\(\)](#) (*in module larray*), 368
[cumprod\(\)](#) (*larray.Array method*), 238
[cumsum\(\)](#) (*larray.Array method*), 237

D

`debug()` (in module *larray*), 451
`degrees()` (in module *larray*), 364
`describe()` (*larray.Array* method), 265
`describe_by()` (*larray.Array* method), 266
`diag()` (in module *larray*), 419
`diff()` (*larray.Array* method), 316
`difference()` (*larray.Axis* method), 147
`difference()` (*larray.IGroup* method), 156
`difference()` (*larray.LGroup* method), 165
`display_names()` (*larray.AxisCollection* property), 172
`divnot0()` (*larray.Array* method), 312
`drop()` (*larray.Array* method), 215
`dtype()` (*larray.Array* property), 210
`dump()` (*larray.Array* method), 394

E

`e` (in module *larray.core.constants*), 461
`edit()` (in module *larray*), 451
`element_equals()` (*larray.Session* method), 435
`empty()` (in module *larray*), 204
`empty_like()` (in module *larray*), 204
`endingwith()` (*larray.Axis* method), 139
`endingwith()` (*larray.IGroup* method), 158
`endingwith()` (*larray.LGroup* method), 167
`eq()` (*larray.Array* method), 287
`equals()` (*larray.Array* method), 285
`equals()` (*larray.Axis* method), 150
`equals()` (*larray.IGroup* method), 154
`equals()` (*larray.LGroup* method), 163
`equals()` (*larray.Session* method), 436
`euler_gamma` (in module *larray.core.constants*), 461
`ExcelReport` (class in *larray*), 399
`exp()` (in module *larray*), 345
`exp2()` (in module *larray*), 347
`expand()` (*larray.Array* method), 276
`expml()` (in module *larray*), 346
`extend()` (*larray.Array* method), 279
`extend()` (*larray.Axis* method), 143
`extend()` (*larray.AxisCollection* method), 181
`eye()` (in module *larray*), 420

F

`fabs()` (in module *larray*), 332
`filter()` (*larray.Array* method), 217
`filter()` (*larray.Session* method), 443
`fix()` (in module *larray*), 344
`floor()` (in module *larray*), 340
`frexp()` (in module *larray*), 378
`from_frame()` (in module *larray*), 411
`from_series()` (in module *larray*), 412
`full()` (in module *larray*), 205

`full_like()` (in module *larray*), 206

G

`get()` (*larray.AxisCollection* method), 178
`get()` (*larray.Session* method), 437
`get_all()` (*larray.AxisCollection* method), 179
`get_by_pos()` (*larray.AxisCollection* method), 179
`get_example_filepath()` (in module *larray*), 413
`get_options()` (in module *larray*), 415
`global_arrays()` (in module *larray*), 430
`graphs_per_row()` (*larray.ExcelReport* property), 403
`graphs_per_row()` (*larray.ReportSheet* property), 407
`growth_rate()` (*larray.Array* method), 264

H

`hypot()` (in module *larray*), 362

I

`i` (*larray.Array* attribute), 211
`i` (*larray.Axis* attribute), 141
`i0()` (in module *larray*), 337
`identity()` (in module *larray*), 419
`ids()` (*larray.AxisCollection* property), 176
`iflat` (*larray.Array* attribute), 213
`ignore_labels()` (*larray.Array* method), 216
`ignore_labels()` (*larray.Axis* method), 148
`IGroup` (class in *larray*), 151
`imag()` (in module *larray*), 375
`index()` (*larray.Axis* method), 138
`index()` (*larray.AxisCollection* method), 175
`indexofmax()` (*larray.Array* method), 307
`indexofmin()` (*larray.Array* method), 305
`indicesofsorted()` (*larray.Array* method), 270
`inf` (in module *larray.core.constants*), 461
`info()` (*larray.Array* property), 208
`info()` (*larray.AxisCollection* property), 174
`insert()` (*larray.Array* method), 280
`insert()` (*larray.Axis* method), 144
`insert()` (*larray.AxisCollection* method), 181
`interp()` (in module *larray*), 328
`intersection()` (*larray.Axis* method), 146
`intersection()` (*larray.IGroup* method), 156
`intersection()` (*larray.LGroup* method), 165
`inverse()` (in module *larray*), 327
`ipfp()` (in module *larray*), 421
`ipoints` (*larray.Array* attribute), 212
`isaxis()` (*larray.AxisCollection* method), 191
`iscompatible()` (*larray.Axis* method), 149
`isin()` (*larray.Array* method), 288
`isinf()` (in module *larray*), 334
`isnan()` (in module *larray*), 333
`items()` (*larray.Array* method), 310

`items()` (*larray.Session method*), 433
`iter_labels()` (*larray.AxisCollection method*), 176

K

`keys()` (*larray.Array method*), 308
`keys()` (*larray.AxisCollection method*), 175
`keys()` (*larray.Session method*), 432

L

`labelofmax()` (*larray.Array method*), 306
`labelofmin()` (*larray.Array method*), 305
`labels()` (*larray.AxisCollection property*), 173
`labels_array()` (*in module larray*), 416
`labelsofsorted()` (*larray.Array method*), 269
`ldexp()` (*in module larray*), 379
`LGroup` (*class in larray*), 160
`load()` (*larray.Session method*), 445
`load_example_data()` (*in module larray*), 430
`local_arrays()` (*in module larray*), 429
`log()` (*in module larray*), 348
`log10()` (*in module larray*), 349
`log1p()` (*in module larray*), 351
`log2()` (*in module larray*), 350
`logaddexp()` (*in module larray*), 352
`logaddexp2()` (*in module larray*), 353
`LSet` (*class in larray*), 169

M

`matching()` (*larray.Axis method*), 140
`matching()` (*larray.LGroup method*), 158
`matching()` (*larray.LGroup method*), 167
`max()` (*larray.Array method*), 301
`max_by()` (*larray.Array method*), 303
`maximum()` (*in module larray*), 324
`mean()` (*larray.Array method*), 239
`mean_by()` (*larray.Array method*), 241
`median()` (*larray.Array method*), 243
`median_by()` (*larray.Array method*), 245
`memory_used()` (*larray.Array property*), 211
`Metadata` (*class in larray*), 380
`min()` (*larray.Array method*), 298
`min_by()` (*larray.Array method*), 299
`minimum()` (*in module larray*), 325

N

`named()` (*larray.LGroup method*), 153
`named()` (*larray.LGroup method*), 162
`names()` (*larray.AxisCollection property*), 172
`names()` (*larray.Session property*), 431
`nan` (*in module larray.core.constants*), 461
`nan_to_num()` (*in module larray*), 335
`nbytes()` (*larray.Array property*), 210
`ndim()` (*larray.Array property*), 209

`ndtest()` (*in module larray*), 200
`new_sheet()` (*larray.ExcelReport method*), 403
`newline()` (*larray.ReportSheet method*), 410
`nonzero()` (*larray.Array method*), 289
`normal()` (*in module larray.random*), 454

O

`ones()` (*in module larray*), 203
`ones_like()` (*in module larray*), 203
`open_excel()` (*in module larray*), 396

P

`percent()` (*larray.Array method*), 262
`percentile()` (*larray.Array method*), 255
`percentile_by()` (*larray.Array method*), 257
`permutation()` (*in module larray.random*), 458
`pi` (*in module larray.core.constants*), 461
`plot()` (*larray.Array property*), 320
`points` (*larray.Array attribute*), 212
`pop()` (*larray.AxisCollection method*), 180
`prepend()` (*larray.Array method*), 277
`prod()` (*larray.Array method*), 233
`prod_by()` (*larray.Array method*), 235
`ptp()` (*larray.Array method*), 259

R

`radians()` (*in module larray*), 365
`randint()` (*in module larray.random*), 453
`ratio()` (*larray.Array method*), 263
`rationot0()` (*larray.Array method*), 263
`read_csv()` (*in module larray*), 382
`read_eurostat()` (*in module larray*), 389
`read_excel()` (*in module larray*), 385
`read_hdf()` (*in module larray*), 388
`read_sas()` (*in module larray*), 389
`read_stata()` (*in module larray*), 389
`read_tsv()` (*in module larray*), 385
`real()` (*in module larray*), 374
`reindex()` (*larray.Array method*), 273
`rename()` (*larray.Array method*), 222
`rename()` (*larray.Axis method*), 143
`rename()` (*larray.AxisCollection method*), 182
`replace()` (*larray.Axis method*), 145
`replace()` (*larray.AxisCollection method*), 183
`ReportSheet` (*class in larray*), 405
`reshape()` (*larray.Array method*), 271
`reshape_like()` (*larray.Array method*), 272
`reverse()` (*larray.Array method*), 227
`rint()` (*in module larray*), 343
`roll()` (*larray.Array method*), 315
`round()` (*in module larray*), 340
`run_editor_on_exception()` (*in module larray*), 452

S

`save()` (*larray.Session* method), 446
`save()` (*larray.Workbook* method), 398
`sequence()` (*in module larray*), 198
Session (*class in larray*), 427
`set()` (*larray.Array* method), 214
`set_axes()` (*larray.Array* method), 220
`set_item_default_size()` (*larray.ExcelReport* method), 402
`set_item_default_size()` (*larray.ReportSheet* method), 407
`set_labels()` (*larray.Array* method), 223
`set_labels()` (*larray.AxisCollection* method), 184
`set_options` (*class in larray*), 414
`shape()` (*larray.Array* property), 209
`shape()` (*larray.AxisCollection* property), 173
`sheet_names()` (*larray.ExcelReport* method), 403
`sheet_names()` (*larray.Workbook* method), 397
`shift()` (*larray.Array* method), 314
`signbit()` (*in module larray*), 376
`sin()` (*in module larray*), 355
`sinc()` (*in module larray*), 338
`sinh()` (*in module larray*), 367
`size()` (*larray.Array* property), 210
`size()` (*larray.AxisCollection* property), 174
`sort_axes()` (*larray.Array* method), 267
`sort_values()` (*larray.Array* method), 268
`split()` (*larray.Axis* method), 148
`split_axes()` (*larray.Array* method), 226
`split_axes()` (*larray.AxisCollection* method), 188
`sqrt()` (*in module larray*), 336
`stack()` (*in module larray*), 417
`startingwith()` (*larray.Axis* method), 139
`startingwith()` (*larray.IGroup* method), 157
`startingwith()` (*larray.LGroup* method), 166
`std()` (*larray.Array* method), 251
`std_by()` (*larray.Array* method), 253
`subaxis()` (*larray.Axis* method), 143
`sum()` (*larray.Array* method), 229
`sum_by()` (*larray.Array* method), 231
`summary()` (*larray.Session* method), 433

T

`tan()` (*in module larray*), 357
`tanh()` (*in module larray*), 369
`template()` (*larray.ExcelReport* property), 402
`template()` (*larray.ReportSheet* property), 406
`template_dir()` (*larray.ExcelReport* property), 401
`template_dir()` (*larray.ReportSheet* property), 406
`to_clipboard()` (*larray.Array* method), 318
`to_csv()` (*larray.Array* method), 390
`to_csv()` (*larray.Session* method), 447
`to_excel()` (*larray.Array* method), 392
`to_excel()` (*larray.ExcelReport* method), 404

`to_excel()` (*larray.Session* method), 448
`to_frame()` (*larray.Array* method), 319
`to_hdf()` (*larray.Array* method), 393
`to_hdf()` (*larray.Axis* method), 150
`to_hdf()` (*larray.IGroup* method), 159
`to_hdf()` (*larray.LGroup* method), 168
`to_hdf()` (*larray.Session* method), 449
`to_pickle()` (*larray.Session* method), 449
`to_series()` (*larray.Array* method), 318
`to_stata()` (*larray.Array* method), 393
`translate()` (*larray.IGroup* method), 155
`translate()` (*larray.LGroup* method), 164
`transpose()` (*larray.Array* method), 275
`transpose()` (*larray.Session* method), 442
`trunc()` (*in module larray*), 342

U

`uniform()` (*in module larray.random*), 456
`union()` (*in module larray*), 416
`union()` (*larray.Axis* method), 146
`union()` (*larray.IGroup* method), 155
`union()` (*larray.LGroup* method), 164
`unique()` (*larray.Array* method), 317
`unwrap()` (*in module larray*), 366
`update()` (*larray.Session* method), 439

V

`values()` (*larray.Array* method), 309
`values()` (*larray.Session* method), 432
`var()` (*larray.Array* method), 247
`var_by()` (*larray.Array* method), 249
`view()` (*in module larray*), 450

W

`where()` (*in module larray*), 323
`with_axis()` (*larray.IGroup* method), 153
`with_axis()` (*larray.LGroup* method), 162
`with_total()` (*larray.Array* method), 261
`without()` (*larray.AxisCollection* method), 186
Workbook (*class in larray*), 397
`wrap_elementwise_array_func()` (*in module larray*), 423

Z

`zeros()` (*in module larray*), 201
`zeros_like()` (*in module larray*), 202
`zip_array_items()` (*in module larray*), 425
`zip_array_values()` (*in module larray*), 424