

---

# LArray Documentation

*Release 0.34.2*

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

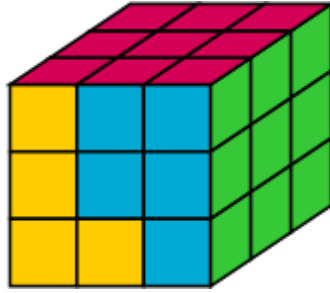
**Oct 23, 2023**



# CONTENTS

<b>1</b>	<b>Library Highlights</b>	<b>3</b>
<b>2</b>	<b>Documentation</b>	<b>5</b>
<b>3</b>	<b>Get in touch</b>	<b>7</b>
<b>4</b>	<b>Contents</b>	<b>9</b>
4.1	Installation . . . . .	9
4.2	Tutorial . . . . .	11
4.3	API Reference . . . . .	109
<b>5</b>	<b>Indices and tables</b>	<b>489</b>
<b>6</b>	<b>Appendix</b>	<b>491</b>
6.1	Change log . . . . .	491
6.2	How to contribute . . . . .	606
	<b>Bibliography</b>	<b>615</b>
	<b>Index</b>	<b>617</b>





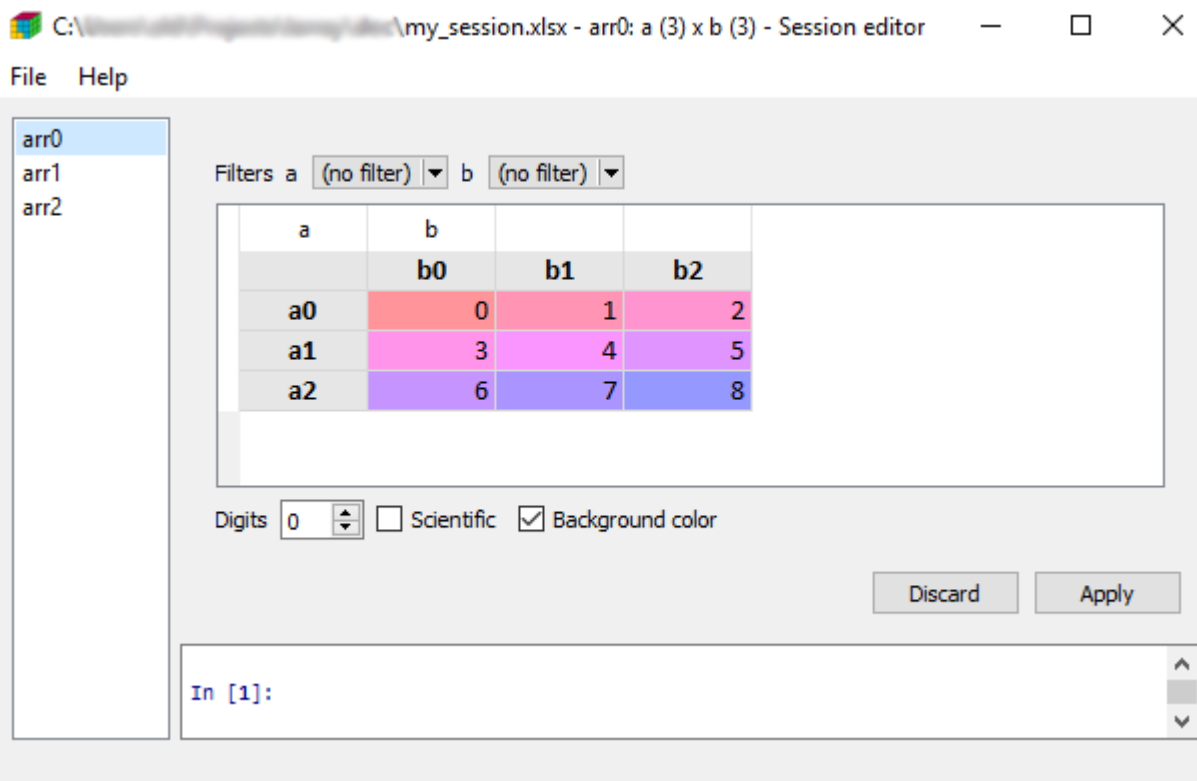
# *LArray*

LArray is an 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 3.8, 3.9, 3.10 or 3.11
- `numpy` (1.22 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.
- `openpyxl`: recommended package for reading and writing Excel 2010 files (ie: .xlsx)
- `xlswriter`: alternative package for writing data, formatting information and, in particular, charts in the Excel 2010 format (ie: .xlsx)
- `xlrd`: for reading data and formatting information from older Excel files (ie: .xls)
- `xlwt`:  
for writing data and formatting information to older Excel files (ie: .xls)
- `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` (version 5): required by *larray-editor* (see below).
- `pyside`: alternative to PyQt.
- `qtpy`: required by *larray-editor*.
- `larray-editor`: required to use the graphical user interface associated with larray. It assumes that *qtpy* and either *pyqt* or *pyside* are installed. On windows, creates also a menu LArray in the Windows Start Menu.

#### For plotting

- `matplotlib`: required for plotting.

## Miscellaneous

- `pydantic`: required to use `CheckedSession`.

### 4.1.5 Update

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

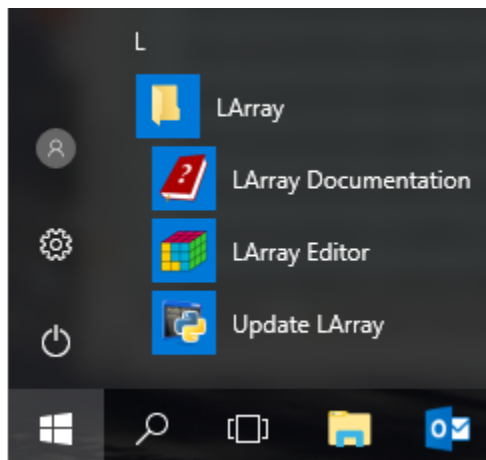
```
conda update larray
```

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 ( $\geq 0.25$ ) 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:

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

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

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

```
[3]: '0.34.2'
```

**Warning:** The tutorial is generated from Jupyter notebooks which work in the “interactive” mode (like in the LArray Editor console). In the interactive mode, there is no need to use the `print()` function to display the content of a variable. Simply writing its name is enough. The same remark applies for the returned value of an expression. In a Python script (file with `.py` extension), you always need to use the `print()` function to display the content of a variable or the value returned by a function or an expression.

```
[4]: s = 1 + 2  
  
     # In the interactive mode, there is no need to use the print() function  
     # to display the content of the variable 's'.  
     # Simply typing 's' is enough  
     s
```

```
[4]: 3
```

```
[5]: # In the interactive mode, there is no need to use the print() function  
     # to display the result of an expression  
     1 + 2
```

```
[5]: 3
```

## 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. There are several ways to create an axis:

```
[6]: # create an axis using one string  
     age = Axis('age=0-9,10-17,18-66,67+')  
     # labels generated using the special syntax start..end  
     time = Axis('time=2015..2017')  
     # labels given as a list  
     gender = Axis(['female', 'male'], 'gender')  
  
     age, gender, time
```

```
[6]: (Axis(['0-9', '10-17', '18-66', '67+'], 'age'),  
     Axis(['female', 'male'], 'gender'),  
     Axis([2015, 2016, 2017], 'time'))
```

**Warning:** When using the string syntax `"axis_name=list,of,labels"` or `"axis_name=start..end"`, LArray will automatically infer the type of labels. For example, `age = Axis("age=0..100")` will create an age axis with labels of type `int`. Mixing numbers with letters or special characters like `+` will create an axis with labels of



type str instead of int. For example, `age = Axis("age=0..98,99+")` will create an age axis with labels of type str instead of int!

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:

```
[7]: # 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
population = Array(data, axes=[age, gender, time])
population
```

```
[7]:  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:

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

# display metadata
population.meta
```

```
[8]: title: population by age, gender and year
      source: Eurostat
```

To get a short summary of an array, type:

```
[9]: # Array summary: metadata + dimensions + description of axes
population.info
```

```
[9]: title: population by age, gender and year
      source: Eurostat
      4 x 2 x 3
      age [4]: '0-9' '10-17' '18-66' '67+'
      gender [2]: 'female' 'male'
```

(continues on next page)

(continued from previous page)

```
time [3]: 2015 2016 2017
dtype: int64
memory used: 192 bytes
```

To get the axes of an array, type:

```
[10]: population.axes
[10]: AxisCollection([
      Axis(['0-9', '10-17', '18-66', '67+'], 'age'),
      Axis(['female', 'male'], 'gender'),
      Axis([2015, 2016, 2017], 'time')
])
```

It is also possible to extract one axis belonging to an array using its name:

```
[11]: # extract the 'time' axis belonging to the 'population' array
time = population.time
time
[11]: Axis([2015, 2016, 2017], 'time')
```

### 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

```
[12]: zeros([age, gender])
[12]: 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
```

```
[13]: ones([age, gender])
[13]: 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
```

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

```
[14]: 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
```

```
[15]: # With initial=1.0 and inc=0.5, we generate the sequence 1.0, 1.5, 2.0, 2.5, 3.0, ...
      sequence(age, initial=1.0, inc=0.5)
```

```
[15]: age  0-9  10-17  18-66  67+
      1.0   1.5   2.0   2.5
```

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

```
[16]: 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`:

```
[17]: # save our population array to a CSV file
      population.to_csv('population_belgium.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`:

```
[18]: population = read_csv('population_belgium.csv')
      population
```

```
[18]: age  gender\time  2015  2016  2017
      0-9      female   633   635   634
      0-9      male    663   665   664
```

(continues on next page)

(continued from previous page)

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

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:

```
[19]: population['67+', 'female', 2017]
```

```
[19]: 1053
```

Labels can be given in arbitrary order:

```
[20]: population[2017, 'female', '67+']
```

```
[20]: 1053
```

When selecting a larger subset the result is an array:

```
[21]: population['female']
```

```
[21]: 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 `[ ]`)

```
[22]: population['female', ['0-9', '10-17']]
```

```
[22]: 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:

```
[23]: # in this case '10-17':'67+' is equivalent to ['10-17', '18-66', '67+']
      population['female', '10-17':'67+']
```

```
[23]: age\time  2015  2016  2017
      10-17   484   486   491
      18-66  3572  3581  3583
      67+    1023  1038  1053
```

```
[24]: # : '18-66' selects all labels between the first one and '18-66'
      # 2017: selects all labels between 2017 and the last one
      population[:, '18-66', 2017:]
```

```
[24]: 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.

---

**Warning:** 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: <somelabel> is ambiguous (valid in <axis1>, <axis2>)`

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

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

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

      immigration
```

```
[25]: 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:

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

```
[26]: 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 population variable:

```
[27]: population
```

```
[27]:  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 ‘gender’ axis using:

```
[28]: population.sum(gender)
```

```
[28]: 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 ‘gender’:

```
[29]: population.sum(age, gender)
```

```
[29]: 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:

```
[30]: population.sum_by(time)
```

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

## Groups

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

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

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

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

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

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

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

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

Still using the same population array:

```
[34]: population
```

```
[34]:  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:

```
[35]: population[working]
```

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

```
[36]: population[nonworking]
```

```
[36]:  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:

```
[37]: population.sum(nonworking)
```

```
[37]: 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 :

```
[38]: population.sum((children, working, nonworking))
```

```
[38]:      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
```

**Warning:** Mixing slices and individual labels inside the [ ] will generate **several groups** (a tuple of groups) instead of a single group. If you want to create a single group using both slices and individual labels, you need to use the `.union()` method (see below).

```
[39]: age_100 = Axis('age=0..100')

# mixing slices and individual labels leads to the creation of several groups (a tuple
↳ of groups)
age_100[0:10, 20, 30, 40]
```

```
[39]: (age[0:10], age[20], age[30], age[40])
```

```
[40]: # the union() method allows to mix slices and individual labels to create a single group
age_100[0:10].union(age_100[20, 30, 40])
```

```
[40]: age[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40].set()
```

## Grouping arrays in a Session

Variables (arrays) may be grouped in *Session* objects. A session is an ordered dict-like container with special I/O methods:

```
[41]: population = zeros([age, gender, time])
births = zeros([age, gender, time])
deaths = zeros([age, gender, time])

# create a session containing the arrays of the model
demography_session = Session(population=population, births=births, deaths=deaths)

# get an array (option 1)
demography_session['population']

# get an array (option 2)
demography_session.births
```

(continues on next page)



(continued from previous page)

```
# modify an array
demography_session.deaths['male'] = 1

# add an array
demography_session.foreigners = zeros([age, gender, time])

# displays names of arrays contained in the session
# (in alphabetical order)
demography_session.names
```

```
[41]: ['births', 'deaths', 'foreigners', 'population']
```

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

```
[42]: # dump all arrays contained in demography_session in one HDF5 file
demography_session.save('demography.h5')
# load all arrays saved in the HDF5 file 'demography.h5' and store them in the 'demography_
↪ session' variable
demography_session = Session('demography.h5')
```

However, development tools like PyCharm do not provide *autocomplete* for objects in Session objects.

*Autocomplete* is the feature in which development tools try to predict the variable or function a user intends to enter after only a few characters have been typed (like word completion in cell phones).

Another way to group objects of a model is to use *CheckedSession*. The *CheckedSession* provide the same methods than *Session* but enable the *autocomplete* feature on objects it contains.

For more details about *Session* and *CheckedSession*, see the *Working With Sessions* section of the tutorial.

To get the list of methods belonging to the *Session* and *CheckedSession* objects, check the *corresponding section* in the API Reference.

## Graphical User Interface (Editor)

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

The `view()` function displays the content of (an) array(s) in a graphical user interface in read-only mode.

For instance, the statement

```
view(population)
```

will open a new window showing the values and axes of the ‘population’ array.

The statement

```
view(demography_session)
```

will show all arrays contained in the ‘demography\_session’.

A session can be directly loaded from a file

```
view('demography.h5')
```

Calling

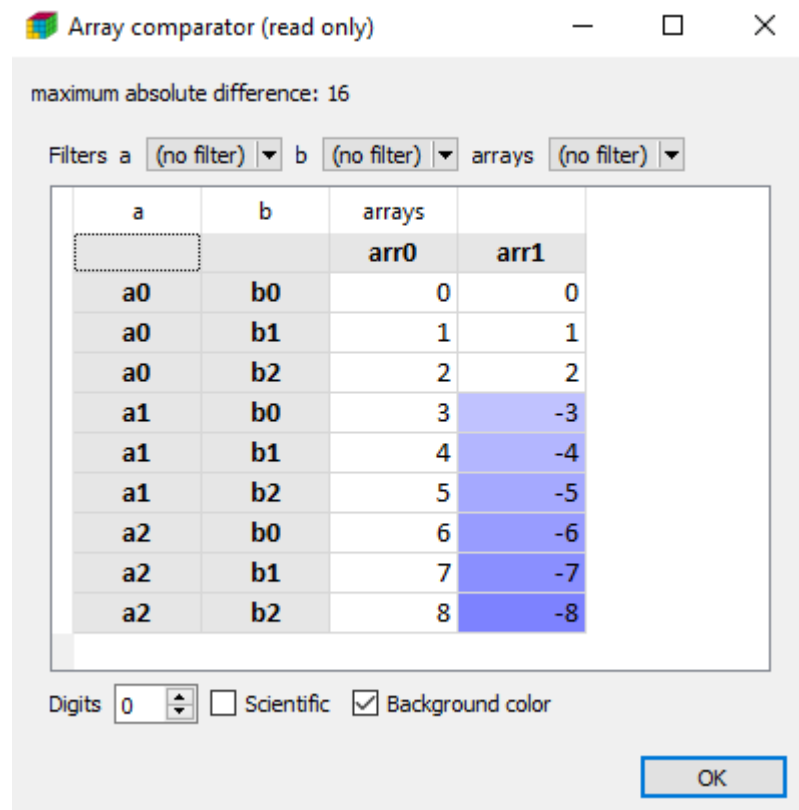
```
view()
```

with no passed argument creates a session with all existing arrays from the current namespace and shows its content.

#### Notes:

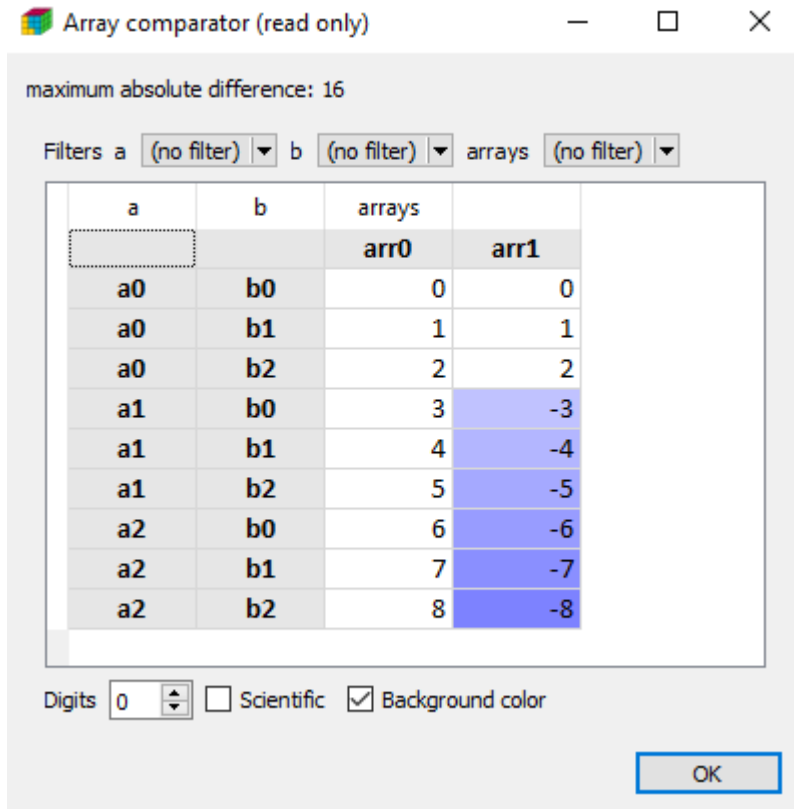
- Calling `view()` will block the execution of the rest of code until the graphical user interface is closed!
- The `larray-editor` tool is automatically available when installing the **larrayenv** metapackage from conda.

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:

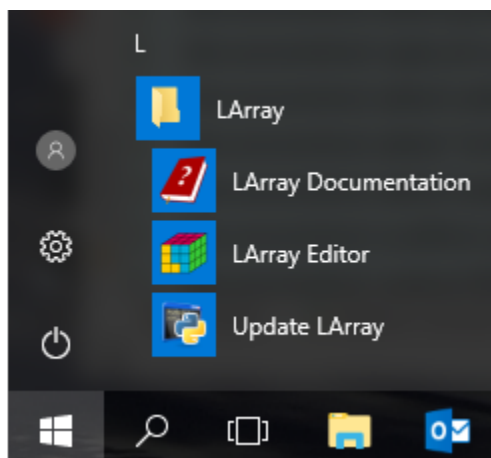
```
arr0 = ndtest((3, 3))
arr1 = ndtest((3, 3))
arr1[['a1', 'a2']] = -arr1[['a1', 'a2']]
compare(arr0, arr1)
```

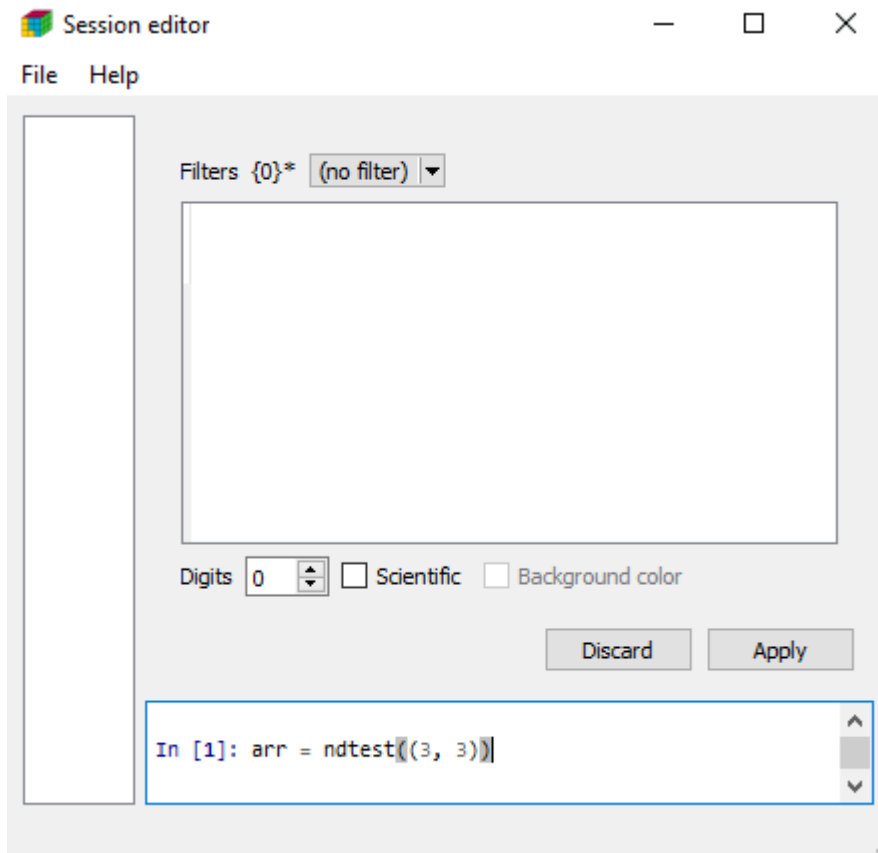


## For Windows Users

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 Load And Dump Arrays

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

```
[2]: 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
```

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
```

```
[3]: 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
```

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 'population' from the sheet 'population'
    population = wb['population'].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')
```

(continues on next page)

(continued from previous page)

```
# 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
```

```
[4]: country gender\time 2013 2014 2015
      Belgium Male 53908 51579 53631
      Belgium Female 55426 53176 56910
      France Male 287410 282381 297028
      France Female 281955 277054 296779
      Germany Male 429645 422225 449512
      Germany Female 464180 446131 475688
```

### 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 population in the file 'population.csv'
      population.to_csv('population.csv')
```

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

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

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')
```

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 'population' and save the array population in it
      population.to_excel('population.xlsx', 'population', overwrite_file=True)
```

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 'population' and dump the array population in it
    wb['population'] = population.dump()
    # add a new sheet 'births' and dump the array births in it
    wb['births'] = births.dump()
```

(continues on next page)

(continued from previous page)

```
# add a new sheet 'deaths' and dump the array deaths in it
wb['deaths'] = deaths.dump()
# 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 population in the file 'population.h5' and associate it with the key
↪'population'
population.to_hdf('population.h5', 'population')
```

### 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
population_BE_FR = read_csv(csv_dir / 'population_narrow_format.csv', wide=False)
population_BE_FR
```

```
[10]: country\time      2013      2014      2015
      Belgium 11137974 11180840 11237274
      France 65600350 66165980 66458153
```

```
[11]: # same for the read_excel function
population_BE_FR = read_excel(filepath_excel, sheet='population_narrow_format',
↪wide=False)
population_BE_FR
```

```
[11]: country\time      2013      2014      2015
      Belgium 11137974 11180840 11237274
      France 65600350 66165980 66458153
```

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 population_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'
      population_BE_FR.to_csv('population_narrow_format.csv', wide=False)

      # same but replace 'value' by 'population'
      population_BE_FR.to_csv('population_narrow_format.csv', wide=False, value_name=
      ↪ 'population')

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

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 population_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['population_narrow_format'] = population_BE_FR.dump(wide=False, value_name=
    ↪ 'population')
    # don't forget to call save()
    wb.save()

# in the sheet 'population_narrow_format', data is written as:
# | country | time | population |
# | ----- | ---- | ----- |
# | 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='population_births_deaths', range='A9:E15')
```

Using `open_excel`, ranges are passed in brackets:

```
with open_excel(filepath_excel) as wb:
    # store sheet 'population_births_deaths' in a temporary variable sh
    sh = wb['population_births_deaths']
    # load the array population from range A1:E7
    population = 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 = 'population_births_deaths'

# save the arrays population, births and deaths in the same sheet 'population_births_and_
# deaths'.
# The 'position' argument is used to shift the location of the second and third arrays to_
# be dumped
population.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 'population_births_deaths' and write 'population' in the first cell
    # note: you can use wb['new_sheet_name'] = "" to create an empty sheet
    wb['population_births_deaths'] = 'population'
    # store sheet 'population_births_deaths' in a temporary variable sh
```

(continues on next page)

(continued from previous page)

```

sh = wb['population_births_deaths']
# dump the array population in sheet 'population_births_deaths' starting at cell A2
sh['A2'] = population.dump()
# add 'births' in cell A10
sh['A10'] = 'births'
# dump the array births in sheet 'population_births_deaths' starting at cell A11
sh['A11'] = births.dump()
# add 'deaths' in cell A19
sh['A19'] = 'deaths'
# dump the array deaths in sheet 'population_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:

```

population_by_country = population.sum('gender')

# export only the data of the array population_by_country starting at cell C2
population_by_country.to_excel('population.xlsx', 'population_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 'population_by_country'
    wb['population_by_country'] = ''

```

(continues on next page)

(continued from previous page)

```

# store sheet 'population_by_country' in a temporary variable sh
sh = wb['population_by_country']
# write extra information (description)
sh['A1'] = 'Population at 1st January by country'
# export column names
sh['A2'] = ['country', 'capital city']
sh['C2'] = population_by_country.time.labels
# export countries as first column
sh['A3'].options(transpose=True).value = population_by_country.country.labels
# export capital cities as second column
sh['B3'].options(transpose=True).value = ['Brussels', 'Paris', 'Berlin']
# export only data of population_by_country
sh['C3'] = population_by_country.dump(header=False)
# or equivalently
sh['C3'] = population_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 'population_missing_axis_name.csv' without
      ↪ using 'nb_axes' argument.
      population = read_csv(csv_dir / 'population_missing_axis_name.csv')
      # shape and data type of the output array are not what we expected
      population.info
```

```
[14]: 6 x 4
      country [6]: 'Belgium' 'Belgium' 'France' 'France' 'Germany' 'Germany'
      {1} [4]: 'gender' '2013' '2014' '2015'
      dtype: object
      memory used: 192 bytes
```

```
[15]: # by setting the 'nb_axes' argument, you can indicate to read_csv the number of axes of
      ↪ the output array
      population = read_csv(csv_dir / 'population_missing_axis_name.csv', nb_axes=3)

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

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

```
[15]: 3 x 2 x 3
      country [3]: 'Belgium' 'France' 'Germany'
      gender [2]: 'Male' 'Female'
      time [3]: 2013 2014 2015
      dtype: int64
      memory used: 144 bytes
```

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

```
[16]: 3 x 2 x 3
      country [3]: 'Belgium' 'France' 'Germany'
      gender [2]: 'Male' 'Female'
      time [3]: 2013 2014 2015
      dtype: int64
      memory used: 144 bytes
```

## 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 / 'population_missing_values.csv')
```

```
[17]: 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
```

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

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

```
[18]: 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
```

```
[19]: # same for the read_excel function
read_excel(filepath_excel, sheet='population_missing_values', fill_value=0)
```

```
[19]: 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
```

## 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 / 'population.csv', sort_rows=True)
```

```
[20]: country  gender\time      2013      2014      2015
Belgium      Female  5665118  5687048  5713206
Belgium      Male    5472856  5493792  5524068
France       Female  33827685 34120851 34283895
France       Male    31772665 32045129 32174258
Germany      Female  41142770 41210540 41362080
Germany      Male    39380976 39556923 39835457
```

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

```
[21]: country gender\time    2013    2014    2015
      Belgium    Female  61235    60841    59713
      Belgium    Male   64371    64173    62561
      France     Female  396581  400607  390526
      France     Male   415762  418721  409145
      Germany    Female  332249  348092  359097
      Germany    Male   349820  366835  378478
```

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

```
[22]: country gender\time    2013    2014    2015
      Belgium    Female  55426    53176    56910
      Belgium    Male   53908    51579    53631
      France     Female  281955  277054  296779
      France     Male   287410  282381  297028
      Germany    Female  464180  446131  475688
      Germany    Male   429645  422225  449512
```

## Metadata (HDF5)

It is possible to add metadata to arrays:

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

```
population.info
```

```
[23]: title: Population at 1st January
      origin: Table demo_jpan from Eurostat
      3 x 2 x 3
      country [3]: 'Belgium' 'France' 'Germany'
      gender [2]: 'Male' 'Female'
      time [3]: 2013 2014 2015
      dtype: int64
      memory used: 144 bytes
```

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

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

new_population = read_hdf('population.h5', 'population')
new_population.info
```

```
[24]: title: Population at 1st January
      origin: Table demo_jpan from Eurostat
      3 x 2 x 3
      country [3]: 'Belgium' 'France' 'Germany'
      gender [2]: 'Male' 'Female'
      time [3]: 2013 2014 2015
      dtype: int64
      memory used: 144 bytes
```

**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.

### 4.2.3 Transforming Arrays (Relabeling, Renaming, Reordering, Sorting, ...)

Import the LArray library:

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

Import the population array from the demography\_eurostat dataset:

```
[2]: demography_eurostat = load_example_data('demography_eurostat')
population = demography_eurostat.population
```

```
# display the 'population' array
population
```

```
[2]: country  gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535
```

### 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:

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

```
[3]: country gender\time      2013      2014      2015      2016      2017
Belgium      Men    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Men    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Men    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535
```

```
[4]: # set all labels of the 'country' axis to uppercase by passing the function str.upper()
population_new_labels = population.set_labels('country', str.upper)
population_new_labels
```

```
[4]: country gender\time      2013      2014      2015      2016      2017
BELGIUM      Male    5472856    5493792    5524068    5569264    5589272
BELGIUM      Female  5665118    5687048    5713206    5741853    5762455
FRANCE       Male    31772665   32045129   32174258   32247386   32318973
FRANCE       Female  33827685   34120851   34283895   34391005   34485148
GERMANY      Male    39380976   39556923   39835457   40514123   40697118
GERMANY      Female  41142770   41210540   41362080   41661561   41824535
```

See [set\\_labels](#) for more details and examples.

## Renaming axes

Rename one axis:

```
[5]: # 'rename' returns a copy of the array
population_new_names = population.rename('time', 'year')
population_new_names
```

```
[5]: country gender\year      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535
```

Rename several axes at once:

```
[6]: population_new_names = population.rename({'gender': 'sex', 'time': 'year'})
population_new_names
```

```
[6]: country sex\year      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
```

(continues on next page)



(continued from previous page)

France	Male	31772665	32045129	32174258	32247386	32318973
France	Female	33827685	34120851	34283895	34391005	34485148
Germany	Male	39380976	39556923	39835457	40514123	40697118
Germany	Female	41142770	41210540	41362080	41661561	41824535

See `rename` for more details and examples.

## Replacing Axes

Replace one axis:

```
[7]: new_gender = Axis('sex=Men,Women')
population_new_axis = population.set_axes('gender', new_gender)
population_new_axis
```

```
[7]: country sex\time      2013      2014      2015      2016      2017
Belgium      Men    5472856    5493792    5524068    5569264    5589272
Belgium      Women  5665118    5687048    5713206    5741853    5762455
France       Men    31772665   32045129   32174258   32247386   32318973
France       Women  33827685   34120851   34283895   34391005   34485148
Germany      Men    39380976   39556923   39835457   40514123   40697118
Germany      Women  41142770   41210540   41362080   41661561   41824535
```

Replace several axes at once:

```
[8]: new_country = Axis('country_codes=BE,FR,DE')
population_new_axes = population.set_axes({'country': new_country, 'gender': new_gender})
population_new_axes
```

```
[8]: country_codes sex\time      2013      2014      2015      2016      2017
      BE      Men    5472856    5493792    5524068    5569264    5589272
      BE      Women  5665118    5687048    5713206    5741853    5762455
      FR      Men    31772665   32045129   32174258   32247386   32318973
      FR      Women  33827685   34120851   34283895   34391005   34485148
      DE      Men    39380976   39556923   39835457   40514123   40697118
      DE      Women  41142770   41210540   41362080   41661561   41824535
```

## 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.

```
[9]: # starting order : country, gender, time
population
```

```
[9]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
```

(continues on next page)

(continued from previous page)

Germany	Male	39380976	39556923	39835457	40514123	40697118
Germany	Female	41142770	41210540	41362080	41661561	41824535

```
[10]: # no argument --> reverse all axes
population_transposed = population.transpose()
```

```
# .T is a shortcut for .transpose()
population_transposed = population.T
```

```
population_transposed
```

```
[10]: time gender\country Belgium    France    Germany
2013      Male  5472856  31772665  39380976
2013      Female 5665118  33827685  41142770
2014      Male  5493792  32045129  39556923
2014      Female 5687048  34120851  41210540
2015      Male  5524068  32174258  39835457
2015      Female 5713206  34283895  41362080
2016      Male  5569264  32247386  40514123
2016      Female 5741853  34391005  41661561
2017      Male  5589272  32318973  40697118
2017      Female 5762455  34485148  41824535
```

```
[11]: # reorder according to list
population_transposed = population.transpose('gender', 'country', 'time')
population_transposed
```

```
[11]: gender country\time    2013    2014    2015    2016    2017
      Male    Belgium  5472856  5493792  5524068  5569264  5589272
      Male    France  31772665  32045129  32174258  32247386  32318973
      Male    Germany 39380976  39556923  39835457  40514123  40697118
      Female Belgium  5665118  5687048  5713206  5741853  5762455
      Female France  33827685  34120851  34283895  34391005  34485148
      Female Germany 41142770  41210540  41362080  41661561  41824535
```

```
[12]: # move 'time' axis at first place
# not mentioned axes come after those which are mentioned (and keep their relative order)
population_transposed = population.transpose('time')
population_transposed
```

```
[12]: time country\gender    Male    Female
2013      Belgium  5472856  5665118
2013      France  31772665  33827685
2013      Germany 39380976  41142770
2014      Belgium  5493792  5687048
2014      France  32045129  34120851
2014      Germany 39556923  41210540
2015      Belgium  5524068  5713206
2015      France  32174258  34283895
2015      Germany 39835457  41362080
2016      Belgium  5569264  5741853
2016      France  32247386  34391005
2016      Germany 40514123  41661561
```

(continues on next page)

(continued from previous page)

2017	Belgium	5589272	5762455
2017	France	32318973	34485148
2017	Germany	40697118	41824535

```
[13]: # move 'gender' axis at last place
      # not mentioned axes come before those which are mentioned (and keep their relative
      # order)
      population_transposed = population.transpose(..., 'gender')
      population_transposed
```

```
[13]: country  time\gender      Male      Female
      Belgium      2013      5472856      5665118
      Belgium      2014      5493792      5687048
      Belgium      2015      5524068      5713206
      Belgium      2016      5569264      5741853
      Belgium      2017      5589272      5762455
      France      2013      31772665      33827685
      France      2014      32045129      34120851
      France      2015      32174258      34283895
      France      2016      32247386      34391005
      France      2017      32318973      34485148
      Germany      2013      39380976      41142770
      Germany      2014      39556923      41210540
      Germany      2015      39835457      41362080
      Germany      2016      40514123      41661561
      Germany      2017      40697118      41824535
```

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

## Dropping Labels

```
[14]: population_labels_dropped = population.drop([2014, 2016])
      population_labels_dropped
```

```
[14]: country  gender\time      2013      2015      2017
      Belgium      Male      5472856      5524068      5589272
      Belgium      Female      5665118      5713206      5762455
      France      Male      31772665      32174258      32318973
      France      Female      33827685      34283895      34485148
      Germany      Male      39380976      39835457      40697118
      Germany      Female      41142770      41362080      41824535
```

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

## Combine And Split Axes

Combine two axes:

```
[15]: population_combined_axes = population.combine_axes(('country', 'gender'))
population_combined_axes
```

```
[15]: country_gender\time      2013      2014      2015      2016      2017
      Belgium_Male  5472856  5493792  5524068  5569264  5589272
      Belgium_Female 5665118  5687048  5713206  5741853  5762455
      France_Male   31772665 32045129 32174258 32247386 32318973
      France_Female 33827685 34120851 34283895 34391005 34485148
      Germany_Male  39380976 39556923 39835457 40514123 40697118
      Germany_Female 41142770 41210540 41362080 41661561 41824535
```

Split an axis:

```
[16]: population_split_axes = population_combined_axes.split_axes('country_gender')
population_split_axes
```

```
[16]: country_gender\time      2013      2014      2015      2016      2017
      Belgium      Male  5472856  5493792  5524068  5569264  5589272
      Belgium      Female 5665118  5687048  5713206  5741853  5762455
      France      Male   31772665 32045129 32174258 32247386 32318973
      France      Female 33827685 34120851 34283895 34391005 34485148
      Germany      Male  39380976 39556923 39835457 40514123 40697118
      Germany      Female 41142770 41210540 41362080 41661561 41824535
```

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:

```
[17]: # reverse years + remove 2013 + add 2018 + copy data for 2017 to 2018
population_new_time = population.reindex('time', '2018..2014', fill_
↪value=population[2017])
population_new_time
```

```
[17]: country_gender\time      2018      2017      2016      2015      2014
      Belgium      Male  5589272  5589272  5569264  5524068  5493792
      Belgium      Female 5762455  5762455  5741853  5713206  5687048
      France      Male   32318973 32318973 32247386 32174258 32045129
      France      Female 34485148 34485148 34391005 34283895 34120851
      Germany      Male  40697118 40697118 40514123 39835457 39556923
      Germany      Female 41824535 41824535 41661561 41362080 41210540
```

or several axes:

```
[18]: population_new = population.reindex({'country': 'country=Luxembourg,Belgium,France,
↪Germany',
                                          'time': 'time=2018..2014'}, fill_value=0)
population_new
```

```
[18]:
```

country	gender\time	2018	2017	2016	2015	2014
Luxembourg	Male	0	0	0	0	0
Luxembourg	Female	0	0	0	0	0
Belgium	Male	0	5589272	5569264	5524068	5493792
Belgium	Female	0	5762455	5741853	5713206	5687048
France	Male	0	32318973	32247386	32174258	32045129
France	Female	0	34485148	34391005	34283895	34120851
Germany	Male	0	40697118	40514123	39835457	39556923
Germany	Female	0	41824535	41661561	41362080	41210540

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

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

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

population_new_country
```

```
[19]:
```

country	gender\time	2013	2014	2015	2016	2017
Belgium	Male	5472856	5493792	5524068	5569264	5589272
Belgium	Female	5665118	5687048	5713206	5741853	5762455
Luxembourg	Male	0	0	0	0	0
Luxembourg	Female	0	0	0	0	0
France	Male	31772665	32045129	32174258	32247386	32318973
France	Female	33827685	34120851	34283895	34391005	34485148
Germany	Male	39380976	39556923	39835457	40514123	40697118
Germany	Female	41142770	41210540	41362080	41661561	41824535

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

## Sorting

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

```
[20]: # get a copy of the 'population_benelux' array
population_benelux = demography_eurostat.population_benelux.copy()
population_benelux
```

```
[20]:
```

country	gender\time	2013	2014	2015	2016	2017
Belgium	Male	5472856	5493792	5524068	5569264	5589272
Belgium	Female	5665118	5687048	5713206	5741853	5762455
Luxembourg	Male	268412	275117	281972	289193	296641
Luxembourg	Female	268627	274563	280986	287056	294026
Netherlands	Male	8307339	8334385	8372858	8417135	8475102
Netherlands	Female	8472236	8494904	8527868	8561985	8606405

Sort an axis (alphabetically if labels are strings)

```
[21]: population_sorted = population_benelux.sort_labels('gender')
population_sorted
```

```
[21]:
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Female	5665118	5687048	5713206	5741853	5762455
	Belgium	Male	5472856	5493792	5524068	5569264	5589272
	Luxembourg	Female	268627	274563	280986	287056	294026
	Luxembourg	Male	268412	275117	281972	289193	296641
	Netherlands	Female	8472236	8494904	8527868	8561985	8606405
	Netherlands	Male	8307339	8334385	8372858	8417135	8475102

Give labels which would sort the axis

```
[22]: population_benelux.labelsofsorted('country')
```

```
[22]:
```

	country	gender\time	2013	...	2017
0	Male	Luxembourg	...	Luxembourg	
0	Female	Luxembourg	...	Luxembourg	
1	Male	Belgium	...	Belgium	
1	Female	Belgium	...	Belgium	
2	Male	Netherlands	...	Netherlands	
2	Female	Netherlands	...	Netherlands	

Sort according to values

```
[23]: population_sorted = population_benelux.sort_values(('Male', 2017))
population_sorted
```

```
[23]:
```

	country	gender\time	2013	2014	2015	2016	2017
	Luxembourg	Male	268412	275117	281972	289193	296641
	Luxembourg	Female	268627	274563	280986	287056	294026
	Belgium	Male	5472856	5493792	5524068	5569264	5589272
	Belgium	Female	5665118	5687048	5713206	5741853	5762455
	Netherlands	Male	8307339	8334385	8372858	8417135	8475102
	Netherlands	Female	8472236	8494904	8527868	8561985	8606405

## 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.

```
[24]: # get a copy of the 'births' array
births = demography_eurostat.births.copy()

# align the two arrays with the 'inner' join method
population_aligned, births_aligned = population_benelux.align(births, join='inner')
```

```
[25]: print('population_benelux before align:')
print(population_benelux)
print()
print('population_benelux after align:')
print(population_aligned)
```

```

population_benelux before align:
  country gender\time      2013      2014      2015      2016      2017
  Belgium      Male  5472856  5493792  5524068  5569264  5589272
  Belgium      Female 5665118  5687048  5713206  5741853  5762455
  Luxembourg    Male  268412  275117  281972  289193  296641
  Luxembourg    Female 268627  274563  280986  287056  294026
  Netherlands    Male  8307339  8334385  8372858  8417135  8475102
  Netherlands    Female 8472236  8494904  8527868  8561985  8606405

population_benelux after align:
  country gender\time      2013      2014      2015      2016      2017
  Belgium      Male  5472856.0  5493792.0  5524068.0  5569264.0  5589272.0
  Belgium      Female 5665118.0  5687048.0  5713206.0  5741853.0  5762455.0

```

```

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

```

```

births before align:
  country gender\time      2013      2014      2015      2016      2017
  Belgium      Male  64371  64173  62561  62428  61179
  Belgium      Female 61235  60841  59713  59468  58511
  France      Male  415762  418721  409145  401388  394058
  France      Female 396581  400607  390526  382937  375987
  Germany      Male  349820  366835  378478  405587  402517
  Germany      Female 332249  348092  359097  386554  382384

births after align:
  country gender\time      2013      2014      2015      2016      2017
  Belgium      Male  64371.0  64173.0  62561.0  62428.0  61179.0
  Belgium      Female 61235.0  60841.0  59713.0  59468.0  58511.0

```

Aligned arrays can then be used in arithmetic operations:

```

[27]: population_aligned - births_aligned

[27]: country gender\time      2013      2014      2015      2016      2017
  Belgium      Male  5408485.0  5429619.0  5461507.0  5506836.0  5528093.0
  Belgium      Female 5603883.0  5626207.0  5653493.0  5682385.0  5703944.0

```

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

## 4.2.4 Combining arrays

Import the LArray library:

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

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

# load 'gender' and 'time' axes
gender = demography_eurostat.gender
time = demography_eurostat.time
```

```
[3]: # load the 'population' array from the 'demography_eurostat' dataset
population = demography_eurostat.population

# show 'population' array
population
```

```
[3]: country  gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535
```

```
[4]: # load the 'population_benelux' array from the 'demography_eurostat' dataset
population_benelux = demography_eurostat.population_benelux

# show 'population_benelux' array
population_benelux
```

```
[4]: country  gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
Luxembourg   Male    268412     275117     281972     289193     296641
Luxembourg   Female  268627     274563     280986     287056     294026
Netherlands  Male    8307339    8334385    8372858    8417135    8475102
Netherlands  Female  8472236    8494904    8527868    8561985    8606405
```

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 a new axis.



## Insert

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

```
# insert new countries before 'France'
```

```
population_new_countries = population.insert(other_countries, before='France')
population_new_countries
```

```
[5]:
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Male	5472856	5493792	5524068	5569264	5589272
	Belgium	Female	5665118	5687048	5713206	5741853	5762455
	Luxembourg	Male	0	0	0	0	0
	Luxembourg	Female	0	0	0	0	0
	Netherlands	Male	0	0	0	0	0
	Netherlands	Female	0	0	0	0	0
	France	Male	31772665	32045129	32174258	32247386	32318973
	France	Female	33827685	34120851	34283895	34391005	34485148
	Germany	Male	39380976	39556923	39835457	40514123	40697118
	Germany	Female	41142770	41210540	41362080	41661561	41824535

```
[6]: # insert new countries after 'France'
```

```
population_new_countries = population.insert(other_countries, after='France')
population_new_countries
```

```
[6]:
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Male	5472856	5493792	5524068	5569264	5589272
	Belgium	Female	5665118	5687048	5713206	5741853	5762455
	France	Male	31772665	32045129	32174258	32247386	32318973
	France	Female	33827685	34120851	34283895	34391005	34485148
	Luxembourg	Male	0	0	0	0	0
	Luxembourg	Female	0	0	0	0	0
	Netherlands	Male	0	0	0	0	0
	Netherlands	Female	0	0	0	0	0
	Germany	Male	39380976	39556923	39835457	40514123	40697118
	Germany	Female	41142770	41210540	41362080	41661561	41824535

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

## Append

Append one element to an axis of an array:

```
[7]: # append data for 'Luxembourg'
```

```
population_new = population.append('country', population_benelux['Luxembourg'],
    ↪ 'Luxembourg')
population_new
```

```
[7]:
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Male	5472856	5493792	5524068	5569264	5589272
	Belgium	Female	5665118	5687048	5713206	5741853	5762455
	France	Male	31772665	32045129	32174258	32247386	32318973
	France	Female	33827685	34120851	34283895	34391005	34485148
	Germany	Male	39380976	39556923	39835457	40514123	40697118

(continues on next page)

(continued from previous page)

Germany	Female	41142770	41210540	41362080	41661561	41824535
Luxembourg	Male	268412	275117	281972	289193	296641
Luxembourg	Female	268627	274563	280986	287056	294026

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

```
[8]: population_lux = stack({'Male': -1, 'Female': 1}, gender)
population_lux
```

```
[8]: gender Male Female
      -1      1
```

```
[9]: population_new = population.append('country', population_lux, 'Luxembourg')
population_new
```

```
[9]: country gender\time    2013    2014    2015    2016    2017
      Belgium      Male  5472856  5493792  5524068  5569264  5589272
      Belgium      Female 5665118  5687048  5713206  5741853  5762455
      France       Male  31772665 32045129 32174258 32247386 32318973
      France       Female 33827685 34120851 34283895 34391005 34485148
      Germany      Male  39380976 39556923 39835457 40514123 40697118
      Germany      Female 41142770 41210540 41362080 41661561 41824535
      Luxembourg   Male      -1      -1      -1      -1      -1
      Luxembourg   Female      1      1      1      1      1
```

The value being appended can also have the axis along which we are appending:

```
[10]: population_nelux = population_benelux[['Netherlands', 'Luxembourg']]
population_nelux
```

```
[10]: country gender\time    2013    2014    2015    2016    2017
      Netherlands      Male  8307339  8334385  8372858  8417135  8475102
      Netherlands      Female 8472236  8494904  8527868  8561985  8606405
      Luxembourg      Male  268412   275117   281972   289193   296641
      Luxembourg      Female 268627   274563   280986   287056   294026
```

```
[11]: population_extended = population.append('country', population_nelux)
population_extended
```

```
[11]: country gender\time    2013    2014    2015    2016    2017
      Belgium      Male  5472856  5493792  5524068  5569264  5589272
      Belgium      Female 5665118  5687048  5713206  5741853  5762455
      France       Male  31772665 32045129 32174258 32247386 32318973
      France       Female 33827685 34120851 34283895 34391005 34485148
      Germany      Male  39380976 39556923 39835457 40514123 40697118
      Germany      Female 41142770 41210540 41362080 41661561 41824535
      Netherlands      Male  8307339  8334385  8372858  8417135  8475102
      Netherlands      Female 8472236  8494904  8527868  8561985  8606405
      Luxembourg      Male  268412   275117   281972   289193   296641
      Luxembourg      Female 268627   274563   280986   287056   294026
```

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

## Prepend

Prepend one element to an axis of an array:

```
[12]: # append data for 'Luxembourg'
population_new = population.prepend('country', population_benelux['Luxembourg'],
↪ 'Luxembourg')
population_new
```

```
[12]:   country  gender\time      2013      2014      2015      2016      2017
Luxembourg      Male    268412    275117    281972    289193    296641
Luxembourg      Female  268627    274563    280986    287056    294026
  Belgium      Male    5472856    5493792    5524068    5569264    5589272
  Belgium      Female  5665118    5687048    5713206    5741853    5762455
  France      Male    31772665    32045129    32174258    32247386    32318973
  France      Female  33827685    34120851    34283895    34391005    34485148
  Germany      Male    39380976    39556923    39835457    40514123    40697118
  Germany      Female  41142770    41210540    41362080    41661561    41824535
```

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

## Stack

Stack several arrays together to create an entirely new dimension

```
[13]: # imagine you have loaded data for each country in different arrays
# (e.g. loaded from different Excel sheets)
population_be = population['Belgium']
population_fr = population['France']
population_de = population['Germany']

print(population_be)
print(population_fr)
print(population_de)
```

```
gender\time      2013      2014      2015      2016      2017
  Male    5472856    5493792    5524068    5569264    5589272
  Female  5665118    5687048    5713206    5741853    5762455
gender\time      2013      2014      2015      2016      2017
  Male    31772665    32045129    32174258    32247386    32318973
  Female  33827685    34120851    34283895    34391005    34485148
gender\time      2013      2014      2015      2016      2017
  Male    39380976    39556923    39835457    40514123    40697118
  Female  41142770    41210540    41362080    41661561    41824535
```

```
[14]: # create a new array with an extra axis 'country' by stacking the three arrays population_
↪ be/fr/de
population_stacked = stack({'Belgium': population_be, 'France': population_fr, 'Germany':
↪ population_de}, 'country')
population_stacked
```

```
[14]: gender  time\country  Belgium  France  Germany
  Male      2013    5472856    31772665    39380976
  Male      2014    5493792    32045129    39556923
```

(continues on next page)

(continued from previous page)

Male	2015	5524068	32174258	39835457
Male	2016	5569264	32247386	40514123
Male	2017	5589272	32318973	40697118
Female	2013	5665118	33827685	41142770
Female	2014	5687048	34120851	41210540
Female	2015	5713206	34283895	41362080
Female	2016	5741853	34391005	41661561
Female	2017	5762455	34485148	41824535

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

## 4.2.5 Indexing, Selecting and Assigning

Import the LArray library:

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

Import the test array population:

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

country	gender\time	2013	2014	2015	2016	2017
Belgium	Male	5472856	5493792	5524068	5569264	5589272
Belgium	Female	5665118	5687048	5713206	5741853	5762455
France	Male	31772665	32045129	32174258	32247386	32318973
France	Female	33827685	34120851	34283895	34391005	34485148
Germany	Male	39380976	39556923	39835457	40514123	40697118
Germany	Female	41142770	41210540	41362080	41661561	41824535

### 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]: population['Belgium', 'Female', 2017]
```

```
[3]: 5762455
```

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
population['Female', 2017, 'Belgium']
```

```
[4]: 5762455
```

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

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

```
[5]: country gender\time      2014      2015      2016
      Belgium      Male    5493792    5524068    5569264
      Belgium      Female  5687048    5713206    5741853
      Germany      Male   39556923   39835457   40514123
      Germany      Female  41210540   41362080   41661561
```

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
      population[2015:]
```

```
[6]: country gender\time      2015      2016      2017
      Belgium      Male    5524068    5569264    5589272
      Belgium      Female  5713206    5741853    5762455
      France      Male   32174258   32247386   32318973
      France      Female  34283895   34391005   34485148
      Germany      Male   39835457   40514123   40697118
      Germany      Female  41362080   41661561   41824535
```

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

```
[7]: 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
```

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

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

```
[8]: country gender\time      2014      2016
      Belgium      Male    5493792    5569264
      Belgium      Female  5687048    5741853
      France      Male   32045129   32247386
      France      Female  34120851   34391005
      Germany      Male   39556923   40514123
      Germany      Female  41210540   41661561
```

**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
```

```
[9]:
```

	country	citizenship	gender\time	2013	2014	2015	2016	2017
	Belgium	Belgium	Male	8822	10512	11378	11055	11082
	Belgium	Belgium	Female	5727	6301	6486	6560	6454
	Belgium	Luxembourg	Male	102	117	105	130	110
	Belgium	Luxembourg	Female	117	123	114	108	118
	Belgium	Netherlands	Male	4185	4222	4183	4199	4138
	Belgium	Netherlands	Female	3737	3844	3942	3664	3632
	Luxembourg	Belgium	Male	896	937	880	762	781
	Luxembourg	Belgium	Female	574	655	622	558	575
	Luxembourg	Luxembourg	Male	694	722	660	740	650
	Luxembourg	Luxembourg	Female	607	586	535	591	549
	Luxembourg	Netherlands	Male	160	165	147	141	167
	Luxembourg	Netherlands	Female	92	97	85	94	119
	Netherlands	Belgium	Male	1063	1141	1113	1364	1493
	Netherlands	Belgium	Female	980	1071	1181	1340	1449
	Netherlands	Luxembourg	Male	23	43	59	70	83
	Netherlands	Luxembourg	Female	24	34	46	60	97
	Netherlands	Netherlands	Male	19374	20037	21119	22707	23750
	Netherlands	Netherlands	Female	16945	17411	18084	19815	20894

```
[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)
```

```
ValueError : 'Belgium' is ambiguous, it is valid in the following axes:
country [3]: 'Belgium' 'Luxembourg' 'Netherlands'
citizenship [3]: 'Belgium' 'Luxembourg' 'Netherlands'
```

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

```
[11]:
```

gender\time	2013	2014	2015	2016	2017
Male	4185	4222	4183	4199	4138
Female	3737	3844	3942	3664	3632

## 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:

- `population[X.time[2015:]]`
- `population.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']]
```

```
[12]: gender\time  2013  2014  2015  2016  2017
      Male  4185  4222  4183  4199  4138
      Female 3737  3844  3942  3664  3632
```

## 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
population[X.time.i[-1]]
```

```
[13]: country\gender      Male      Female
      Belgium  5589272  5762455
      France   32318973  34485148
      Germany  40697118  41824535
```

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

```
[14]: country gender\time      2015      2016      2017
      Belgium Male  5524068  5569264  5589272
      Belgium Female 5713206  5741853  5762455
      France Male 32174258 32247386 32318973
      France Female 34283895 34391005 34485148
      Germany Male 39835457 40514123 40697118
      Germany Female 41362080 41661561 41824535
```

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

```
[15]: country gender\time      2013      2015      2017
      Belgium Male  5472856  5524068  5589272
      Belgium Female 5665118  5713206  5762455
```

(continues on next page)

(continued from previous page)

France	Male	31772665	32174258	32318973
France	Female	33827685	34283895	34485148
Germany	Male	39380976	39835457	40697118
Germany	Female	41142770	41362080	41824535

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

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

```
# with labels
population[X.time[:year]]
```

```
[16]: 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
```

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

```
[17]: country gender\time      2013      2014
Belgium      Male    5472856    5493792
Belgium      Female  5665118    5687048
France       Male    31772665    32045129
France       Female  33827685    34120851
Germany      Male    39380976    39556923
Germany      Female  41142770    41210540
```

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
population.i[0, :, -3:]
```

```
[18]: gender\time      2015      2016      2017
      Male    5524068    5569264    5589272
      Female  5713206    5741853    5762455
```



## Using Groups In Selections

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

```
population[even_years]
```

```
[19]: country  gender\time      2014      2016
      Belgium    Male    5493792    5569264
      Belgium    Female  5687048    5741853
      France     Male   32045129    32247386
      France     Female  34120851    34391005
      Germany    Male   39556923    40514123
      Germany    Female  41210540    41661561
```

## Boolean Filtering

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

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

```
[20]: country  gender\time      2014      2016
      Belgium    Male    5493792    5569264
      Belgium    Female  5687048    5741853
      France     Male   32045129    32247386
      France     Female  34120851    34391005
      Germany    Male   39556923    40514123
      Germany    Female  41210540    41661561
```

or data:

```
[21]: # select population for the year 2017
      population_2017 = population[2017]

      # select all data with a value greater than 30 million
      population_2017[population_2017 > 30e6]
```

```
[21]: country_gender  France_Male  France_Female  Germany_Male  Germany_Female
              32318973      34485148      40697118      41824535
```

---

**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=population.country)
      start_year
```

```
[22]: country  Belgium  France  Germany
              2015     2016     2017
```

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

```

-----
KeyError                                Traceback (most recent call last)
Cell In[23], line 1
----> 1 population[X.time >= start_year]

File ~/checkouts/readthedocs.org/user_builds/larray/envs/stable/lib/python3.11/site-
packages/larray/core/expr.py:13, in ExprNode._binop.<locals>.opmethod(self, other)
    10 def opmethod(self, other):
    11     # evaluate eagerly when possible
    12     if isinstance(other, ABCArray):
--> 13         self_value = self.evaluate(other.axes)
    14         return getattr(self_value, f'__{opname}__')(other)
    15     else:

File ~/checkouts/readthedocs.org/user_builds/larray/envs/stable/lib/python3.11/site-
packages/larray/core/axis.py:3780, in AxisReference.evaluate(self, context)
    3773 def evaluate(self, context) -> Axis:
    3774     r"""
    3775     Parameters
    3776     -----
    3777     context : AxisCollection
    3778         Use axes from this collection
    3779     """
-> 3780     return context[self.name]

File ~/checkouts/readthedocs.org/user_builds/larray/envs/stable/lib/python3.11/site-
packages/larray/core/axis.py:1744, in AxisCollection.__getitem__(self, key)
    1742     return self._map[key]
    1743 else:
-> 1744     raise KeyError(f"axis '{key}' not found in {self}")

KeyError: "axis 'time' not found in {country}"

```

## Iterating over an axis

Iterating over an axis is straightforward:

```
[24]: for year in population.time:
      print(year)
```

```

2013
2014
2015
2016
2017

```

## Assigning subsets

### Assigning A Value

Assigning a value to a subset is simple:

```
[25]: population[2017] = 0
population
```

```
[25]: country gender\time      2013      2014      2015      2016  2017
Belgium      Male    5472856    5493792    5524068    5569264    0
Belgium      Female  5665118    5687048    5713206    5741853    0
France       Male    31772665   32045129   32174258   32247386    0
France       Female  33827685   34120851   34283895   34391005    0
Germany      Male    39380976   39556923   39835457   40514123    0
Germany      Female  41142770   41210540   41362080   41661561    0
```

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

```
[26]: # store the data associated with the year 2016 in a new variable
population_2016 = population[2016]
population_2016
```

```
[26]: country\gender      Male      Female
      Belgium    5569264    5741853
      France    32247386    34391005
      Germany    40514123    41661561
```

```
[27]: # now, we modify the new variable
population_2016['Belgium'] = 0

# and we can see that the original array has been also modified
population
```

```
[27]: country gender\time      2013      2014      2015      2016  2017
Belgium      Male    5472856    5493792    5524068    0    0
Belgium      Female  5665118    5687048    5713206    0    0
France       Male    31772665   32045129   32174258   32247386    0
France       Female  33827685   34120851   34283895   34391005    0
Germany      Male    39380976   39556923   39835457   40514123    0
Germany      Female  41142770   41210540   41362080   41661561    0
```

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:

```
[28]: # reload the 'population' array
population = load_example_data('demography_eurostat').population

# create a second 'population2' variable
population2 = population
population2
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Male	5472856	5493792	5524068	5569264	5589272
	Belgium	Female	5665118	5687048	5713206	5741853	5762455
	France	Male	31772665	32045129	32174258	32247386	32318973
	France	Female	33827685	34120851	34283895	34391005	34485148
	Germany	Male	39380976	39556923	39835457	40514123	40697118
	Germany	Female	41142770	41210540	41362080	41661561	41824535

```
[29]: # set all data corresponding to the year 2017 to 0
population2[2017] = 0
population2
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Male	5472856	5493792	5524068	5569264	0
	Belgium	Female	5665118	5687048	5713206	5741853	0
	France	Male	31772665	32045129	32174258	32247386	0
	France	Female	33827685	34120851	34283895	34391005	0
	Germany	Male	39380976	39556923	39835457	40514123	0
	Germany	Female	41142770	41210540	41362080	41661561	0

```
[30]: # and now take a look of what happened to the original array 'population'
# after modifying the 'population2' array
population
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Male	5472856	5493792	5524068	5569264	0
	Belgium	Female	5665118	5687048	5713206	5741853	0
	France	Male	31772665	32045129	32174258	32247386	0
	France	Female	33827685	34120851	34283895	34391005	0
	Germany	Male	39380976	39556923	39835457	40514123	0
	Germany	Female	41142770	41210540	41362080	41661561	0

**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.

```
[31]: # reload the 'population' array
population = load_example_data('demography_eurostat').population

# copy the 'population' array and store the copy in a new variable
population2 = population.copy()

# modify the copy
population2[2017] = 0
population2
```

```
[31]: country gender\time      2013      2014      2015      2016  2017
      Belgium      Male  5472856  5493792  5524068  5569264    0
      Belgium      Female 5665118  5687048  5713206  5741853    0
      France       Male  31772665 32045129 32174258 32247386    0
      France       Female 33827685 34120851 34283895 34391005    0
      Germany      Male  39380976 39556923 39835457 40514123    0
      Germany      Female 41142770 41210540 41362080 41661561    0
```

```
[32]: # the data from the original array have not been modified
      population
```

```
[32]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male  5472856  5493792  5524068  5569264  5589272
      Belgium      Female 5665118  5687048  5713206  5741853  5762455
      France       Male  31772665 32045129 32174258 32247386 32318973
      France       Female 33827685 34120851 34283895 34391005 34485148
      Germany      Male  39380976 39556923 39835457 40514123 40697118
      Germany      Female 41142770 41210540 41362080 41661561 41824535
```

## 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.

```
[33]: # select population for the year 2015
      population_2015 = population[2015]

      # propagate population for the year 2015 to all next years
      population[2016:] = population_2015

      population
```

```
[33]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male  5472856  5493792  5524068  5524068  5524068
      Belgium      Female 5665118  5687048  5713206  5713206  5713206
      France       Male  31772665 32045129 32174258 32174258 32174258
      France       Female 33827685 34120851 34283895 34283895 34283895
      Germany      Male  39380976 39556923 39835457 39835457 39835457
      Germany      Female 41142770 41210540 41362080 41362080 41362080
```

**Warning:** The array being assigned must have compatible axes (i.e. same axes names and same labels) with the target subset.

```
[34]: # replace 'Male' and 'Female' labels by 'M' and 'F'
      population_2015 = population_2015.set_labels('gender', 'M,F')
      population_2015
```

```
[34]: country\gender      M      F
      Belgium  5524068  5713206
      France   32174258 34283895
      Germany  39835457 41362080
```

```
[35]: # now let's try to repeat the assignment operation above with the new labels.
# An error is raised because of incompatible axes
try:
    population[2016:] = population_2015
except Exception as e:
    print(type(e).__name__, ': ', e)

ValueError : incompatible axes:
Axis(['Male', 'Female'], 'gender')
vs
Axis(['M', 'F'], 'gender')
```

## 4.2.6 Arithmetic Operations

Import the LArray library:

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

Load the population array from the demography\_eurostat dataset:

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

# extract the 'country', 'gender' and 'time' axes
country = demography_eurostat.country
gender = demography_eurostat.gender
time = demography_eurostat.time

# extract the 'population' array
population = demography_eurostat.population

# show the 'population' array
population
```

```
[2]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535
```

### Basics

One can do all usual arithmetic operations on an array, it will apply the operation to all elements individually

```
[3]: # 'true' division
population_in_millions = population / 1_000_000
population_in_millions
```

```
[3]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male     5.472856    5.493792    5.524068    5.569264    5.589272
```

(continues on next page)

(continued from previous page)

Belgium	Female	5.665118	5.687048	5.713206	5.741853	5.762455
France	Male	31.772665	32.045129	32.174258	32.247386	32.318973
France	Female	33.827685	34.120851	34.283895	34.391005	34.485148
Germany	Male	39.380976	39.556923	39.835457	40.514123	40.697118
Germany	Female	41.14277	41.21054	41.36208	41.661561	41.824535

```
[4]: # 'floor' division
population_in_millions = population // 1_000_000
population_in_millions
```

```
[4]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male      5      5      5      5      5
Belgium      Female    5      5      5      5      5
France       Male     31     32     32     32     32
France       Female    33     34     34     34     34
Germany      Male     39     39     39     40     40
Germany      Female    41     41     41     41     41
```

**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).

```
[5]: # % means modulo (aka remainder of division)
population % 1_000_000
```

```
[5]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male  472856 493792 524068 569264 589272
Belgium      Female 665118 687048 713206 741853 762455
France       Male  772665 45129 174258 247386 318973
France       Female 827685 120851 283895 391005 485148
Germany      Male  380976 556923 835457 514123 697118
Germany      Female 142770 210540 362080 661561 824535
```

```
[6]: # ** means raising to the power
print(ndtest(4))
ndtest(4) ** 3
```

```
a  a0  a1  a2  a3
   0   1   2   3
```

```
[6]: a  a0  a1  a2  a3
     0   1   8  27
```

More interestingly, binary operators as above also works between two arrays.

Let us imagine a rate of population growth which is constant over time but different by gender and country:

```
[7]: growth_rate = Array(data=[[1.011, 1.010], [1.013, 1.011], [1.010, 1.009]], axes=[country,
↪ gender])
growth_rate
```

```
[7]: country\gender    Male    Female
      Belgium  1.011    1.01
      France   1.013    1.011
      Germany   1.01    1.009
```

```
[8]: # we store the population of the year 2017 in a new variable
      population_2017 = population[2017]
      population_2017
```

```
[8]: country\gender    Male    Female
      Belgium  5589272    5762455
      France   32318973   34485148
      Germany  40697118   41824535
```

```
[9]: # perform an arithmetic operation between two arrays
      population_2018 = population_2017 * growth_rate
      population_2018
```

```
[9]: country\gender    Male    Female
      Belgium  5650753.992    5820079.55
      France   32739119.648999996  34864484.628
      Germany  41104089.18    42200955.815
```

---

**Note:** Be careful when mixing different data types. You can use the method `astype` to change the data type of an array.

---

```
[10]: # force the resulting matrix to be an integer matrix
      population_2018 = (population_2017 * growth_rate).astype(int)
      population_2018
```

```
[10]: country\gender    Male    Female
      Belgium  5650753    5820079
      France   32739119   34864484
      Germany  41104089   42200955
```

### 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

```
[11]: # let's change the order of axes of the 'constant_growth_rate' array
      transposed_growth_rate = growth_rate.transpose()

      # look at the order of the new 'transposed_growth_rate' array:
      # 'gender' is the first axis while 'country' is the second
      transposed_growth_rate
```

```
[11]: gender\country  Belgium  France  Germany
      Male    1.011    1.013    1.01
      Female    1.01    1.011    1.009
```



```
[12]: # look at the order of the 'population_2017' array:
      # 'country' is the first axis while 'gender' is the second
      population_2017
```

```
[12]: country\gender      Male      Female
      Belgium  5589272    5762455
      France   32318973   34485148
      Germany  40697118   41824535
```

```
[13]: # LArray doesn't care of axes order when performing
      # arithmetic operations between arrays
      population_2018 = population_2017 * transposed_growth_rate
      population_2018
```

```
[13]: country\gender      Male      Female
      Belgium  5650753.992    5820079.55
      France   32739119.648999996  34864484.628
      Germany  41104089.18    42200955.815
```

### Axes must be compatible

Arithmetic operations between two arrays only works when they have compatible axes (i.e. same list of labels in the same order).

```
[14]: # show 'population_2017'
      population_2017
```

```
[14]: country\gender      Male      Female
      Belgium  5589272    5762455
      France   32318973   34485148
      Germany  40697118   41824535
```

### Order of labels matters

```
[15]: # let us imagine that the labels of the 'country' axis
      # of the 'constant_growth_rate' array are in a different order
      # than in the 'population_2017' array
      reordered_growth_rate = growth_rate.reindex('country', ['Germany', 'Belgium', 'France'])
      reordered_growth_rate
```

```
[15]: country\gender      Male      Female
      Germany  1.01    1.009
      Belgium  1.011    1.01
      France   1.013    1.011
```

```
[16]: # when doing arithmetic operations,
      # the order of labels counts
      try:
          population_2018 = population_2017 * reordered_growth_rate
      except Exception as e:
          print(type(e).__name__, e)
```

```
ValueError incompatible axes:
Axis(['Germany', 'Belgium', 'France'], 'country')
vs
Axis(['Belgium', 'France', 'Germany'], 'country')
```

### No extra or missing labels are permitted

```
[17]: # let us imagine that the 'country' axis of
# the 'constant_growth_rate' array has an extra
# label 'Netherlands' compared to the same axis of
# the 'population_2017' array
growth_rate_netherlands = Array([1.012, 1.], population.gender)
growth_rate_extra_country = growth_rate.append('country', growth_rate_netherlands, label=
↪ 'Netherlands')
growth_rate_extra_country
```

```
[17]: country\gender    Male    Female
      Belgium    1.011      1.01
      France    1.013      1.011
      Germany    1.01      1.009
      Netherlands 1.012      1.0
```

```
[18]: # when doing arithmetic operations,
# no extra or missing labels are permitted
try:
    population_2018 = population_2017 * growth_rate_extra_country
except Exception as e:
    print(type(e).__name__, e)
```

```
ValueError incompatible axes:
Axis(['Belgium', 'France', 'Germany', 'Netherlands'], 'country')
vs
Axis(['Belgium', 'France', 'Germany'], 'country')
```

### Ignoring labels (risky)

**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 and **SHOULD NEVER BEEN USED IN A MODEL**. Use this method only for quick tests or rapid data exploration.

```
[19]: # let us imagine that the labels of the 'country' axis
# of the 'constant_growth_rate' array are the
# country codes instead of the country full names
growth_rate_country_codes = growth_rate.set_labels('country', ['BE', 'FR', 'DE'])
growth_rate_country_codes
```

```
[19]: country\gender    Male    Female
      BE    1.011    1.01
      FR    1.013    1.011
      DE    1.01    1.009
```

```
[20]: # use the .ignore_labels() method on axis 'country'
      # to avoid the incompatible axes error (risky)
      population_2018 = population_2017 * growth_rate_country_codes.ignore_labels('country')
      population_2018
```

```
[20]: country\gender          Male          Female
      Belgium          5650753.992          5820079.55
      France    32739119.648999996    34864484.628
      Germany          41104089.18    42200955.815
```

### Extra Or Missing Axes (Broadcasting)

The condition that axes must be compatible only applies on common axes. Making arithmetic operations between two arrays having the same axes is intuitive. However, arithmetic operations between two arrays can be performed even if the second array has extra and/or missing axes compared to the first one. Such mechanism is called **broadcasting**. It allows to make a lot of arithmetic operations without using any loop. This is a great advantage since using loops in Python can be highly time consuming (especially nested loops) and should be avoided as much as possible.

To understand how broadcasting works, let us start with a simple example. We assume we have the population of both men and women cumulated for each country:

```
[21]: population_by_country = population_2017['Male'] + population_2017['Female']
      population_by_country
```

```
[21]: country    Belgium    France    Germany
      11351727    66804121    82521653
```

We also assume we have the proportion of each gender in the population and that proportion is supposed to be the same for all countries:

```
[22]: gender_proportion = Array([0.49, 0.51], gender)
      gender_proportion
```

```
[22]: gender    Male    Female
      0.49    0.51
```

Using the two 1D arrays above, we can naively compute the population by country and gender as follow:

```
[23]: # define a new variable with both 'country' and 'gender' axes to store the result
      population_by_country_and_gender = zeros([country, gender], dtype=int)

      # loop over the 'country' and 'gender' axes
      for c in country:
          for g in gender:
              population_by_country_and_gender[c, g] = population_by_country[c] * gender_
              ↪proportion[g]

      # display the result
      population_by_country_and_gender
```

```
[23]: country\gender      Male      Female
      Belgium    5562346    5789380
      France    32734019    34070101
      Germany   40435609    42086043
```

Relying on the broadcasting mechanism, the calculation above becomes:

```
[24]: # the outer product is done automatically.
      # No need to use any loop -> saves a lot of computation time
      population_by_country_and_gender = population_by_country * gender_proportion

      # display the result
      population_by_country_and_gender.astype(int)
```

```
[24]: country\gender      Male      Female
      Belgium    5562346    5789380
      France    32734019    34070101
      Germany   40435609    42086043
```

In the calculation above, LArray automatically creates a resulting array with axes given by the union of the axes of the two arrays involved in the arithmetic operation.

Let us do the same calculation but we add a common time axis:

```
[25]: population_by_country_and_year = population['Male'] + population['Female']
      population_by_country_and_year
```

```
[25]: country\time      2013      2014      2015      2016      2017
      Belgium  11137974  11180840  11237274  11311117  11351727
      France   65600350  66165980  66458153  66638391  66804121
      Germany   80523746  80767463  81197537  82175684  82521653
```

```
[26]: gender_proportion_by_year = Array([[0.49, 0.485, 0.495, 0.492, 0.498],
      [0.51, 0.515, 0.505, 0.508, 0.502]], [gender, time])
      gender_proportion_by_year
```

```
[26]: gender\time  2013   2014   2015   2016   2017
      Male    0.49  0.485  0.495  0.492  0.498
      Female  0.51  0.515  0.505  0.508  0.502
```

Without the broadcasting mechanism, the computation of the population by country, gender and year would have been:

```
[27]: # define a new variable to store the result.
      # Its axes is the union of the axes of the two arrays
      # involved in the arithmetic operation
      population_by_country_gender_year = zeros([country, gender, time], dtype=int)

      # loop over axes which are not present in both arrays
      # involved in the arithmetic operation
      for c in country:
          for g in gender:
              # all subsets below have the same 'time' axis
              population_by_country_gender_year[c, g] = population_by_country_and_year[c] *
      ↪ gender_proportion_by_year[g]
```

(continues on next page)

(continued from previous page)

population\_by\_country\_gender\_year

```
[27]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male  5457607  5422707  5562450  5565069  5653160
      Belgium      Female 5680366  5758132  5674823  5746047  5698566
      France       Male  32144171 32090500 32896785 32786088 33268452
      France       Female 33456178 34075479 33561367 33852302 33535668
      Germany      Male  39456635 39172219 40192780 40430436 41095783
      Germany      Female 41067110 41595243 41004756 41745247 41425869
```

Once again, the above calculation can be simplified as:

```
[28]: # No need to use any loop -> saves a lot of computation time
      population_by_country_gender_year = population_by_country_and_year * gender_proportion_
      ↪by_year

      # display the result
      population_by_country_gender_year.astype(int)
```

```
[28]: country time\gender      Male      Female
      Belgium      2013  5457607  5680366
      Belgium      2014  5422707  5758132
      Belgium      2015  5562450  5674823
      Belgium      2016  5565069  5746047
      Belgium      2017  5653160  5698566
      France       2013  32144171  33456178
      France       2014  32090500  34075479
      France       2015  32896785  33561367
      France       2016  32786088  33852302
      France       2017  33268452  33535668
      Germany      2013  39456635  41067110
      Germany      2014  39172219  41595243
      Germany      2015  40192780  41004756
      Germany      2016  40430436  41745247
      Germany      2017  41095783  41425869
```

**Warning:** Broadcasting is a powerful mechanism but can be confusing at first. It can lead to unexpected results. In particular, if axes which are supposed to be common are not, you will get a resulting array with extra axes you didn't want.

For example, imagine that the name of the time axis is time for the first array but period for the second:

```
[29]: gender_proportion_by_year = gender_proportion_by_year.rename('time', 'period')
      gender_proportion_by_year
```

```
[29]: gender\period  2013   2014   2015   2016   2017
      Male    0.49  0.485  0.495  0.492  0.498
      Female  0.51  0.515  0.505  0.508  0.502
```

```
[30]: population_by_country_and_year
```

```
[30]: country\time      2013      2014      2015      2016      2017
      Belgium 11137974 11180840 11237274 11311117 11351727
      France  65600350 66165980 66458153 66638391 66804121
      Germany 80523746 80767463 81197537 82175684 82521653
```

```
[31]: # the two arrays below have a "time" axis with two different names: 'time' and 'period'.
      # LArray will treat the "time" axis of the two arrays as two different "time" axes
      population_by_country_gender_year = population_by_country_and_year * gender_proportion_
      ↪by_year
```

```
# as a consequence, the result of the multiplication of the two arrays is not what we_
↪expected
```

```
population_by_country_gender_year.astype(int)
```

```
[31]: country time gender\period      2013      2014      2015      2016      2017
      Belgium 2013      Male  5457607  5401917  5513297  5479883  5546711
      Belgium 2013      Female 5680366  5736056  5624676  5658090  5591262
      Belgium 2014      Male  5478611  5422707  5534515  5500973  5568058
      Belgium 2014      Female 5702228  5758132  5646324  5679866  5612781
      Belgium 2015      Male  5506264  5450077  5562450  5528738  5596162
      Belgium 2015      Female 5731009  5787196  5674823  5708535  5641111
      Belgium 2016      Male  5542447  5485891  5599002  5565069  5632936
      Belgium 2016      Female 5768669  5825225  5712114  5746047  5678180
      Belgium 2017      Male  5562346  5505587  5619104  5585049  5653160
      Belgium 2017      Female 5789380  5846139  5732622  5766677  5698566
      France  2013      Male  32144171 31816169 32472173 32275372 32668974
      France  2013      Female 33456178 33784180 33128176 33324977 32931375
      France  2014      Male  32421330 32090500 32752160 32553662 32950658
      France  2014      Female 33744649 34075479 33413819 33612317 33215321
      France  2015      Male  32564494 32232204 32896785 32697411 33096160
      France  2015      Female 33893658 34225948 33561367 33760741 33361992
      France  2016      Male  32652811 32319619 32986003 32786088 33185918
      France  2016      Female 33985579 34318771 33652387 33852302 33452472
      France  2017      Male  32734019 32399998 33068039 32867627 33268452
      France  2017      Female 34070101 34404122 33736081 33936493 33535668
      Germany 2013      Male  39456635 39054016 39859254 39617683 40100825
      Germany 2013      Female 41067110 41469729 40664491 40906062 40422920
      Germany 2014      Male  39576056 39172219 39979894 39737591 40222196
      Germany 2014      Female 41191406 41595243 40787568 41029871 40545266
      Germany 2015      Male  39786793 39380805 40192780 39949188 40436373
      Germany 2015      Female 41410743 41816731 41004756 41248348 40761163
      Germany 2016      Male  40266085 39855206 40676963 40430436 40923490
      Germany 2016      Female 41909598 42320477 41498720 41745247 41252193
      Germany 2017      Male  40435609 40023001 40848218 40600653 41095783
      Germany 2017      Female 42086043 42498651 41673434 41920999 41425869
```

## 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:

```
[32]: # test which values are greater than 10 millions
population > 10e6
```

```
[32]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male False False False False False
Belgium      Female False False False False False
France       Male  True  True  True  True  True
France       Female True  True  True  True  True
Germany      Male  True  True  True  True  True
Germany      Female True  True  True  True  True
```

Comparison operations can be combined using Python bitwise operators:

Operator	Meaning
&	and
	or
~	not

```
[33]: # test which values are greater than 10 millions and less than 40 millions
(population > 10e6) & (population < 40e6)
```

```
[33]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male False False False False False
Belgium      Female False False False False False
France       Male  True  True  True  True  True
France       Female True  True  True  True  True
Germany      Male  True  True  True  False False
Germany      Female False False False False False
```

```
[34]: # test which values are less than 10 millions or greater than 40 millions
(population < 10e6) | (population > 40e6)
```

```
[34]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male  True  True  True  True  True
Belgium      Female True  True  True  True  True
France       Male False False False False False
France       Female False False False False False
Germany      Male False False False  True  True
Germany      Female True  True  True  True  True
```

```
[35]: # test which values are not less than 10 millions
~(population < 10e6)
```

```
[35]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male False False False False False
Belgium      Female False False False False False
France       Male  True  True  True  True  True
France       Female True  True  True  True  True
Germany      Male  True  True  True  True  True
Germany      Female True  True  True  True  True
```

The returned boolean array can then be used in selections and assignments:

```
[36]: population_copy = population.copy()

# set all values greater than 40 millions to 40 millions
population_copy[population_copy > 40e6] = 40e6
population_copy
```

```
[36]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male 5472856 5493792 5524068 5569264 5589272
Belgium      Female 5665118 5687048 5713206 5741853 5762455
France       Male 31772665 32045129 32174258 32247386 32318973
France       Female 33827685 34120851 34283895 34391005 34485148
Germany      Male 39380976 39556923 39835457 40000000 40000000
Germany      Female 40000000 40000000 40000000 40000000 40000000
```

Boolean operations can be made between arrays:

```
[37]: # test where the two arrays have the same values
population == population_copy
```

```
[37]: country gender\time 2013 2014 2015 2016 2017
Belgium      Male  True  True  True  True  True
Belgium      Female True  True  True  True  True
France       Male  True  True  True  True  True
France       Female True  True  True  True  True
Germany      Male  True  True  True  False False
Germany      Female False False False False False
```

To test if all values between are equals, use the *equals* method:

```
[38]: population.equals(population_copy)
```

```
[38]: False
```



## 4.2.7 Aggregations

Import the LArray library:

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

Load the population array and related axes from the demography\_eurostat dataset:

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

# extract the 'country', 'gender' and 'time' axes
country = demography_eurostat.country
gender = demography_eurostat.gender
time = demography_eurostat.time

# extract the 'population_5_countries' array as 'population'
population = demography_eurostat.population_5_countries

# show the 'population' array
population
```

```
[2]:
```

	country	gender\time	2013	2014	2015	2016	2017
	Belgium	Male	5472856	5493792	5524068	5569264	5589272
	Belgium	Female	5665118	5687048	5713206	5741853	5762455
	France	Male	31772665	32045129	32174258	32247386	32318973
	France	Female	33827685	34120851	34283895	34391005	34485148
	Germany	Male	39380976	39556923	39835457	40514123	40697118
	Germany	Female	41142770	41210540	41362080	41661561	41824535
	Luxembourg	Male	268412	275117	281972	289193	296641
	Luxembourg	Female	268627	274563	280986	287056	294026
	Netherlands	Male	8307339	8334385	8372858	8417135	8475102
	Netherlands	Female	8472236	8494904	8527868	8561985	8606405

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:

```
[3]: population.sum(gender)
```

```
[3]:
```

	country\time	2013	2014	2015	2016	2017
	Belgium	11137974	11180840	11237274	11311117	11351727
	France	65600350	66165980	66458153	66638391	66804121
	Germany	80523746	80767463	81197537	82175684	82521653
	Luxembourg	537039	549680	562958	576249	590667
	Netherlands	16779575	16829289	16900726	16979120	17081507

or several axes (axes are separated by commas ,):

```
[4]: population.sum(country, gender)
```

```
[4]: time      2013      2014      2015      2016      2017
      174578684 175493252 176356648 177680561 178349675
```

Calculate the sum along all axes except one by appending `_by` to the aggregation function:

```
[5]: population.sum_by(time)
```

```
[5]: time      2013      2014      2015      2016      2017
      174578684 175493252 176356648 177680561 178349675
```

Calculate the sum along groups (the groups belonging to the same axis must grouped inside parentheses ()):

```
[6]: benelux = population.country['Belgium', 'Netherlands', 'Luxembourg'] >> 'benelux'
      fr_de = population.country['France', 'Germany'] >> 'FR+DE'
```

```
population.sum((benelux, fr_de))
```

```
[6]: country gender\time      2013      2014      2015      2016      2017
      benelux      Male 14048607 14103294 14178898 14275592 14361015
      benelux      Female 14405981 14456515 14522060 14590894 14662886
      FR+DE      Male 71153641 71602052 72009715 72761509 73016091
      FR+DE      Female 74970455 75331391 75645975 76052566 76309683
```

Mixing axes and groups in aggregations:

```
[7]: population.sum(gender, (benelux, fr_de))
```

```
[7]: country\time      2013      2014      2015      2016      2017
      benelux 28454588 28559809 28700958 28866486 29023901
      FR+DE 146124096 146933443 147655690 148814075 149325774
```

**Warning:** Mixing slices and individual labels inside the `[ ]` will generate **several groups** (a tuple of groups) instead of a single group. If you want to create a single group using both slices and individual labels, you need to use the `.union()` method (see below).

```
[8]: # mixing slices and individual labels leads to the creation of several groups (a tuple
      ↪ of groups)
```

```
except_2016 = time[:2015, 2017]
except_2016
```

```
[8]: (time[:2015], time[2017])
```

```
[9]: # leading to potentially unexpected results
```

```
population.sum(except_2016)
```

```
[9]: country gender\time      :2015      2017
      Belgium      Male 16490716 5589272
      Belgium      Female 17065372 5762455
      France      Male 95992052 32318973
      France      Female 102232431 34485148
      Germany      Male 118773356 40697118
      Germany      Female 123715390 41824535
```

(continues on next page)

(continued from previous page)

Luxembourg	Male	825501	296641
Luxembourg	Female	824176	294026
Netherlands	Male	25014582	8475102
Netherlands	Female	25495008	8606405

```
[10]: # the union() method allows to mix slices and individual labels to create a single group
except_2016 = time[:2015].union(time[2017])
except_2016
```

```
[10]: time[2013, 2014, 2015, 2017].set()
```

```
[11]: population.sum(except_2016)
```

```
[11]: country\gender      Male      Female
      Belgium  22079988  22827827
      France  128311025  136717579
      Germany  159470474  165539925
      Luxembourg  1122142   1118202
      Netherlands  33489684  34101413
```

## 4.2.8 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
[7]: title: Immigration by age group, sex and citizenship
source: table migr_imm1ctz from Eurostat
3 x 3 x 2 x 5
country [3]: 'Belgium' 'Luxembourg' 'Netherlands'
citizenship [3]: 'Belgium' 'Luxembourg' 'Netherlands'
gender [2]: 'Male' 'Female'
time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 360 bytes
```

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
```

(continues on next page)

(continued from previous page)

```
immigration_subset = immigration[X.country['Belgium', 'Netherlands'], 'Female', 2015:]
immigration_subset
```

```
[8]:   country citizenship\time  2015  2016  2017
      Belgium      Belgium  6486  6560  6454
      Belgium      Luxembourg  114   108   118
      Belgium      Netherlands 3942  3664  3632
      Netherlands      Belgium 1181  1340  1449
      Netherlands      Luxembourg  46    60    97
      Netherlands      Netherlands 18084 19815 20894
```

Using the String syntax, the same selection becomes:

```
[9]: immigration_subset = immigration['country[Belgium,Netherlands]', 'Female', 2015:]
immigration_subset
```

```
[9]:   country citizenship\time  2015  2016  2017
      Belgium      Belgium  6486  6560  6454
      Belgium      Luxembourg  114   108   118
      Belgium      Netherlands 3942  3664  3632
      Netherlands      Belgium 1181  1340  1449
      Netherlands      Luxembourg  46    60    97
      Netherlands      Netherlands 18084 19815 20894
```

## Aggregation

An example of an aggregation using the Pythonic syntax is:

```
[10]: immigration.mean((X.time[2014::2] >> 'even_years', X.time[:,2] >> 'odd_years'),
    ↪ 'citizenship')
```

```
[10]:   country gender\time          even_years          odd_years
      Belgium      Male  5039.166666666667  4900.555555555556
      Belgium      Female 3433.3333333333335 3369.6666666666665
      Luxembourg      Male  577.8333333333334  559.4444444444445
      Luxembourg      Female 430.1666666666667  417.5555555555556
      Netherlands      Male  7560.333333333333  7564.111111111111
      Netherlands      Female 6621.833333333333 6633.333333333333
```

Using the String syntax, the same aggregation becomes:

```
[11]: immigration.mean('time[2014::2] >> even_years; time[:,2] >> odd_years', 'citizenship')
```

```
[11]:   country gender\time          even_years          odd_years
      Belgium      Male  5039.166666666667  4900.555555555556
      Belgium      Female 3433.3333333333335 3369.6666666666665
      Luxembourg      Male  577.8333333333334  559.4444444444445
      Luxembourg      Female 430.1666666666667  417.5555555555556
      Netherlands      Male  7560.333333333333  7564.111111111111
      Netherlands      Female 6621.833333333333 6633.333333333333
```

where we used ; to separate groups of labels from the same axis.

## 4.2.9 Plotting

Import the LArray library:

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

Import the test array population from the demography\_eurostat dataset:

```
[2]: demography_eurostat = load_example_data('demography_eurostat')
population = demography_eurostat.population / 1_000_000
```

```
# show the 'population' array
population
```

```
[2]: country  gender\time      2013      2014      2015      2016      2017
Belgium      Male    5.472856  5.493792  5.524068  5.569264  5.589272
Belgium      Female  5.665118  5.687048  5.713206  5.741853  5.762455
France       Male    31.772665 32.045129 32.174258 32.247386 32.318973
France       Female  33.827685 34.120851 34.283895 34.391005 34.485148
Germany      Male    39.380976 39.556923 39.835457 40.514123 40.697118
Germany      Female  41.14277  41.21054  41.36208  41.661561 41.824535
```

Inline matplotlib (required in notebooks):

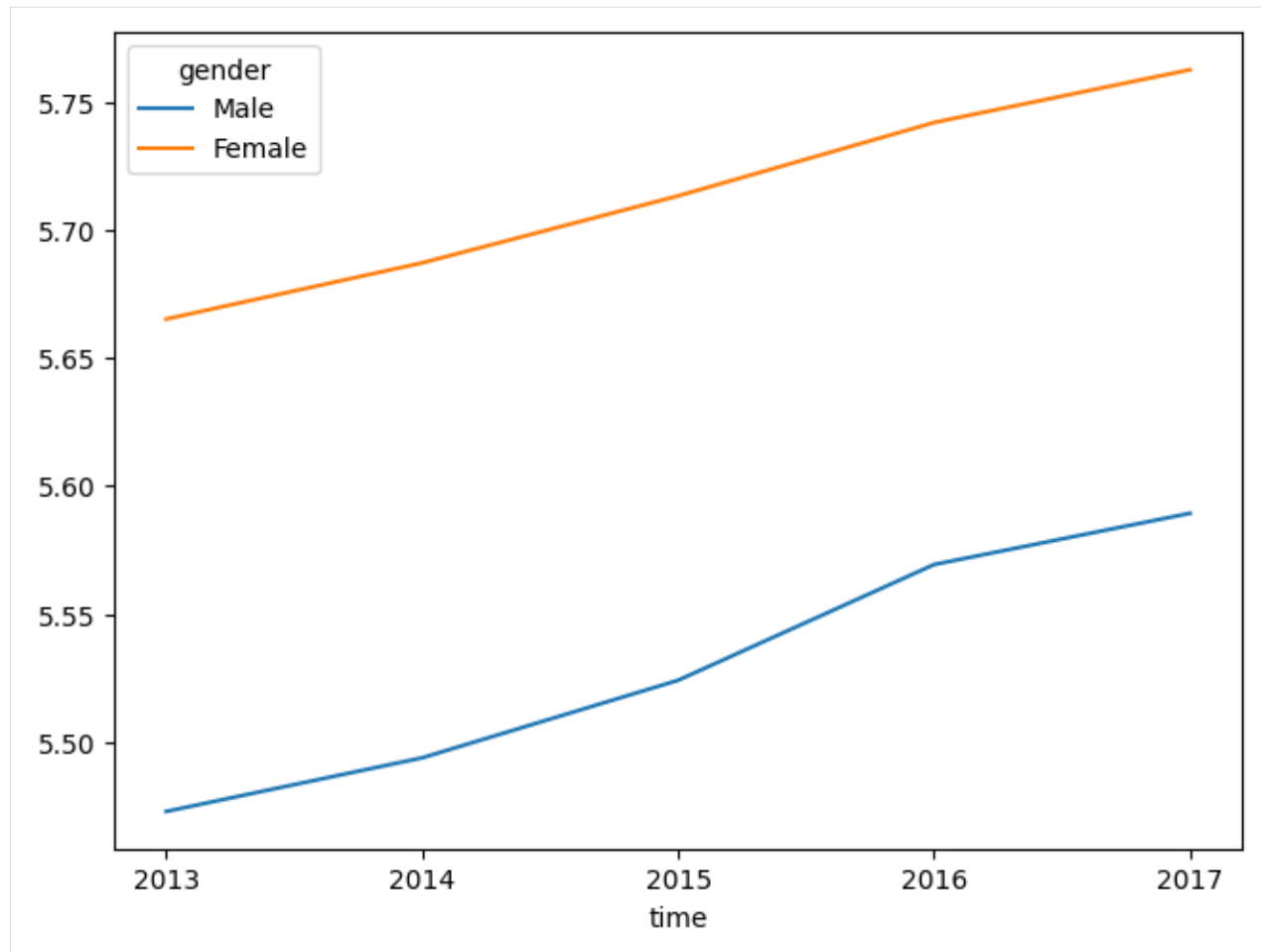
```
[3]: %matplotlib inline
```

In a Python script, add the following import on top of the script:

```
[4]: import matplotlib.pyplot as plt
```

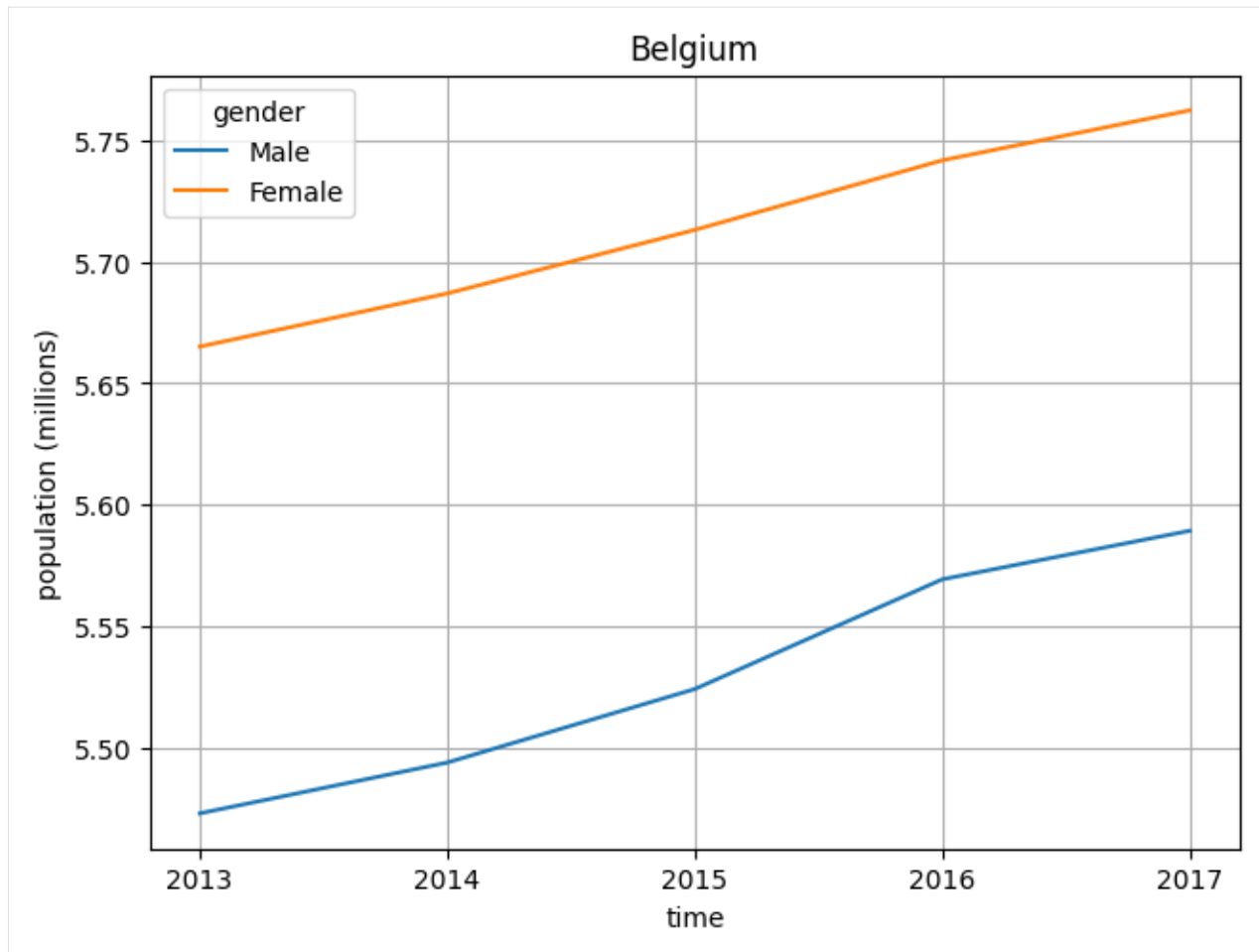
Create and show a simple plot (last axis define the different curves to draw):

```
[5]: population['Belgium'].plot()
# shows the figure
plt.show()
```



- Create a Line plot with grid, user-defined xticks, label and title.
- Save the plot as a png file (using `plt.savefig()`).
- Show the plot:

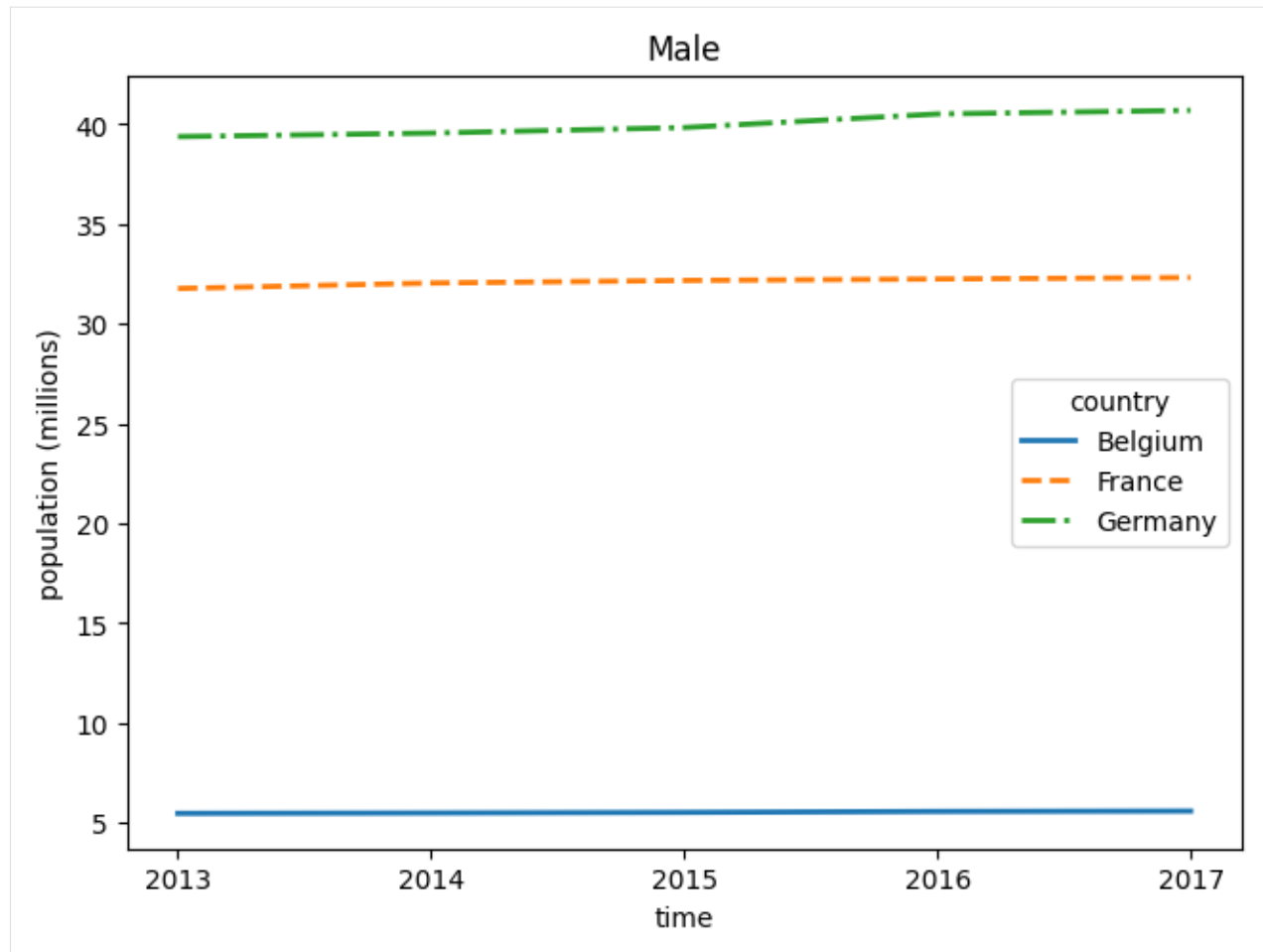
```
[6]: population['Belgium'].plot(grid=True, xticks=population.time, ylabel='population_
    ↳(millions)', title='Belgium')
    # saves figure in a file (see matplotlib.pyplot.savefig documentation for more details)
    plt.savefig('Belgium_population.png')
    # WARNING: show() resets the current figure after showing it! Do not call it before_
    ↳savefig
    plt.show()
```



Specify line styles and width:

```
[7]: # line styles: '-' for solid line, '--' for dashed line, '-.' for dash-dotted line and ':'
    ↪ for dotted line
population['Male'].plot(style=['-', '--', '-.', ':'], linewidth=2,
                        xticks=population.time, ylabel='population (millions)', title=
    ↪ 'Male')
plt.show()
```

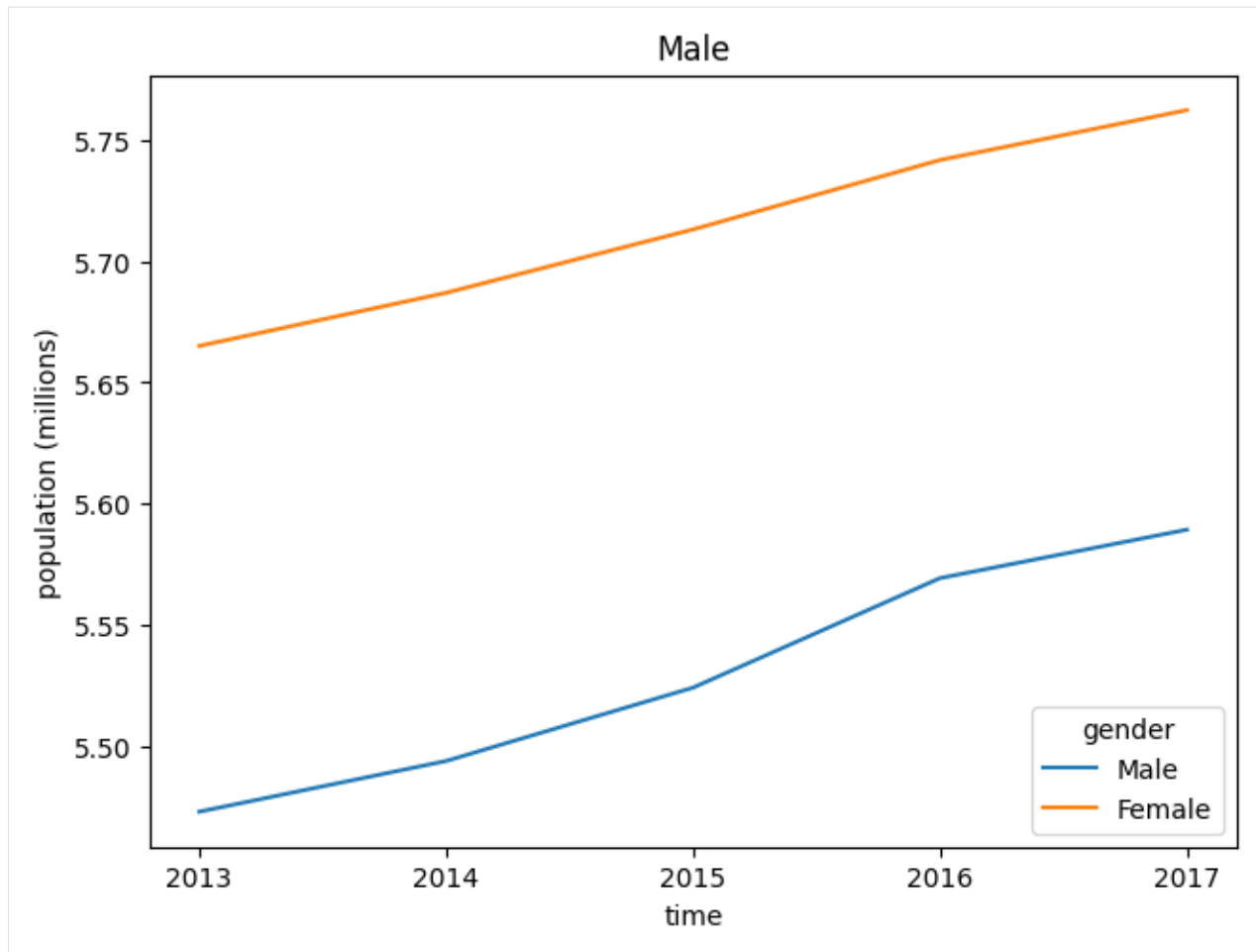




Configuring the legend can be done by passing a dict to the legend argument. For example, to put the legend in a specific position inside the graph, one would use `legend={'loc': <position>}`.

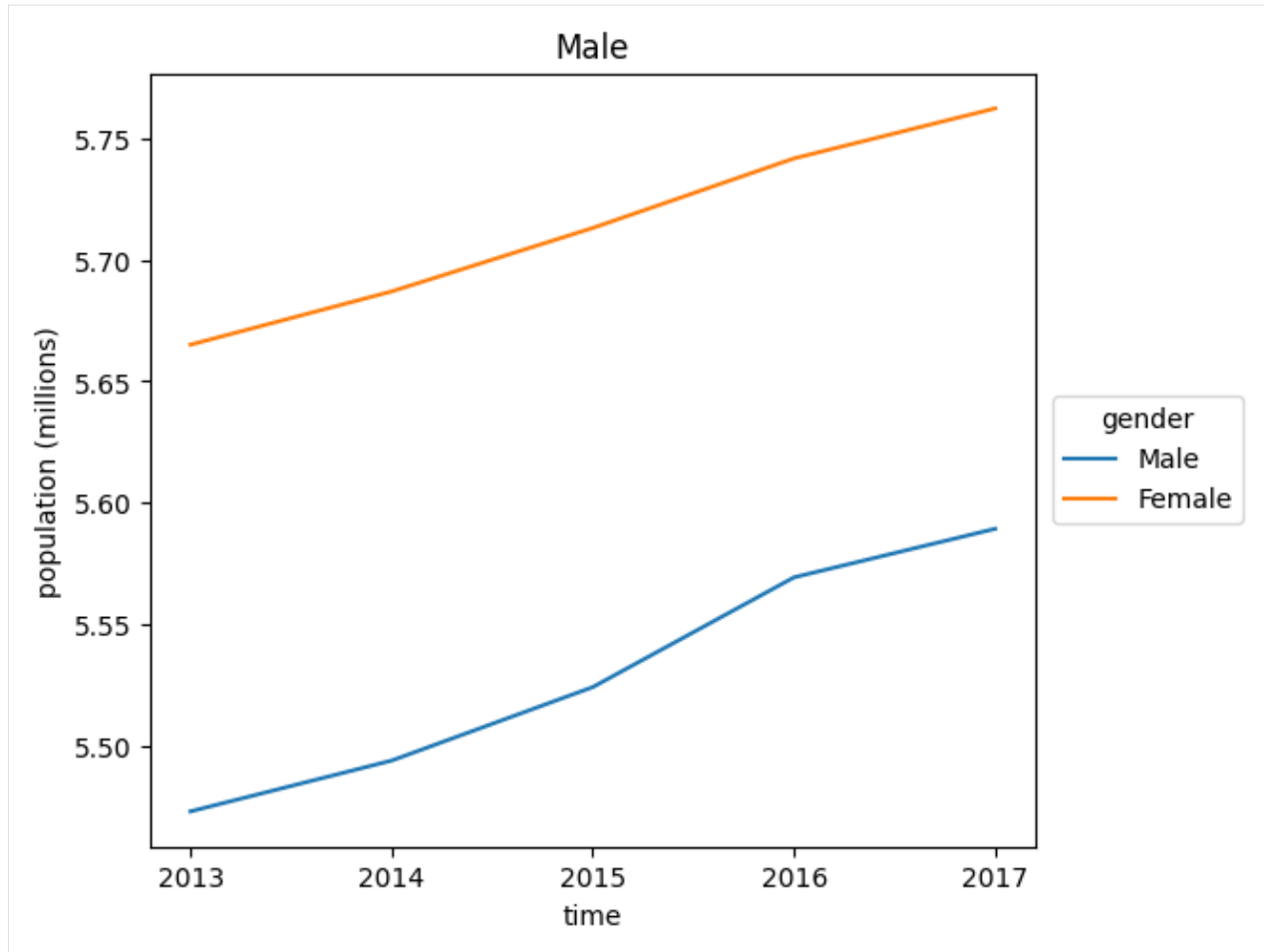
Where <position> can be: 'best' (default), 'upper right', 'upper left', 'lower left', 'lower right', 'right', 'center left', 'center right', 'lower center', 'upper center' or 'center'.

```
[8]: population['Belgium'].plot(xticks=population.time, ylabel='population (millions)', title=
    ↪ 'Male', legend={'loc': 'lower right'})
    plt.show()
```



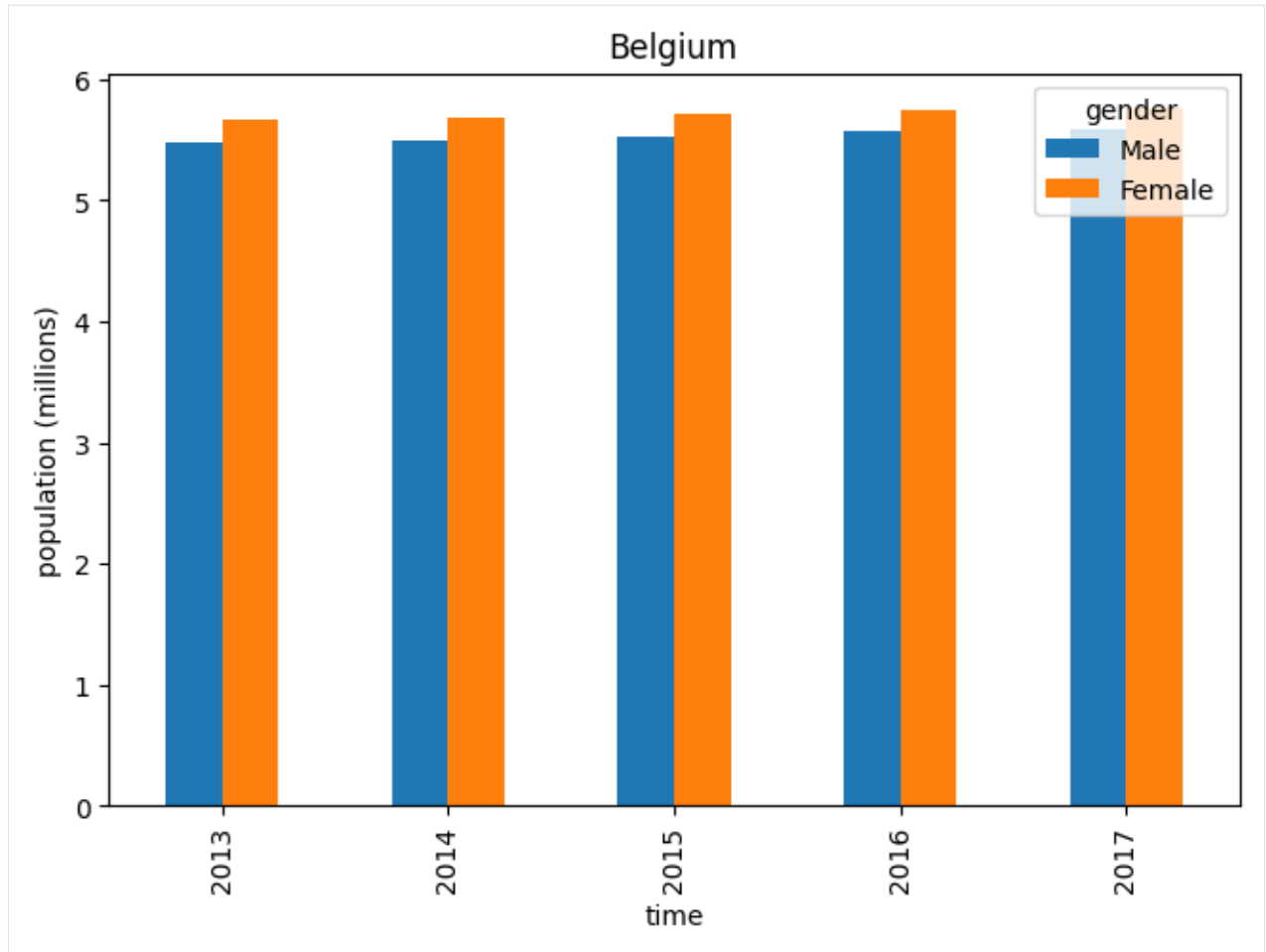
There are many other ways to customize the legend, see the “Other parameters” section of [matplotlib’s legend documentation](#). For example, to put the legend outside the plot:

```
[9]: population['Belgium'].plot(xticks=population.time, ylabel='population (millions)', title=
    ↪ 'Male',
                                legend={'bbox_to_anchor': (1.25, 0.6)})
plt.show()
```



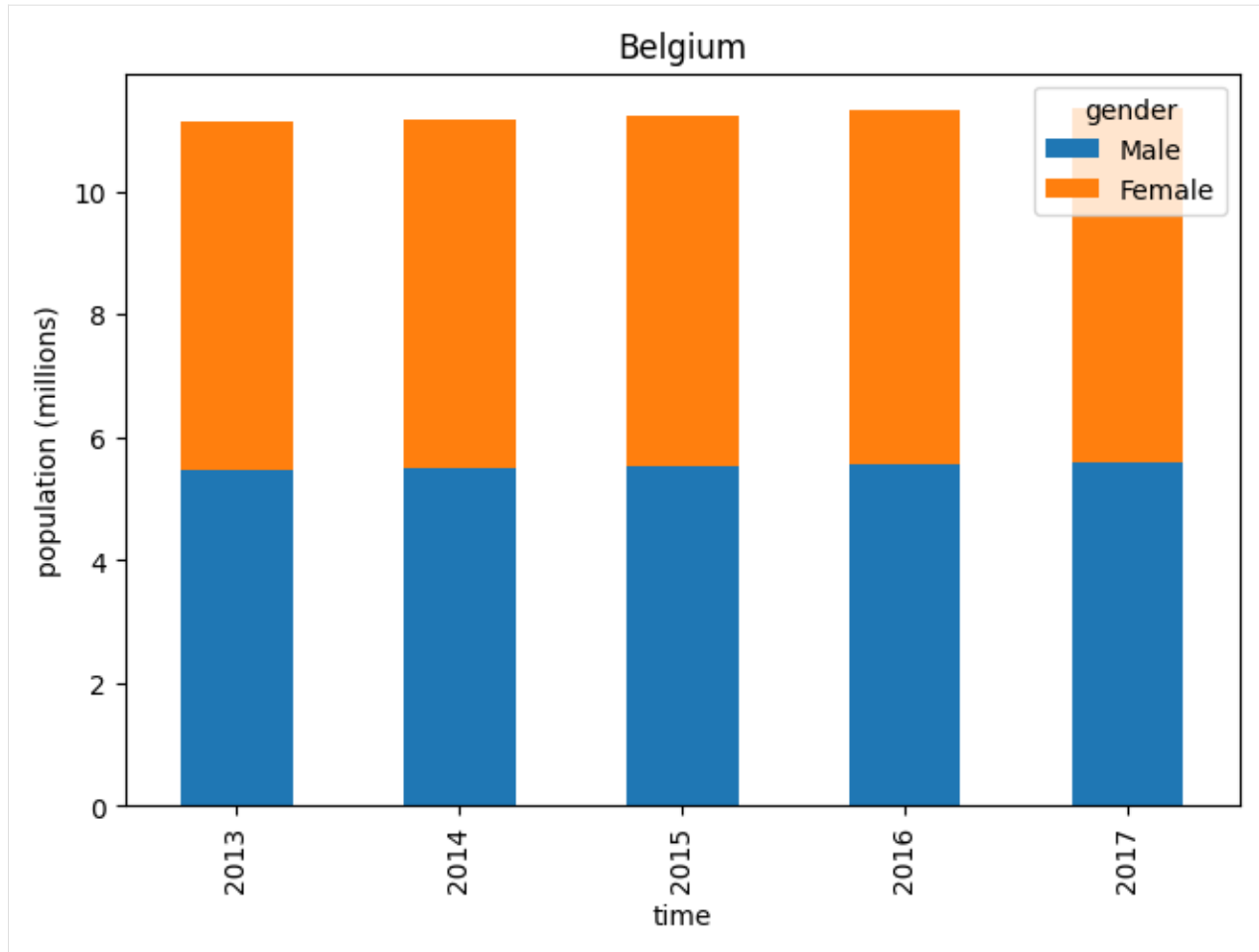
Create a Bar plot:

```
[10]: population['Belgium'].plot.bar(ylabel='population (millions)', title='Belgium')
plt.show()
```



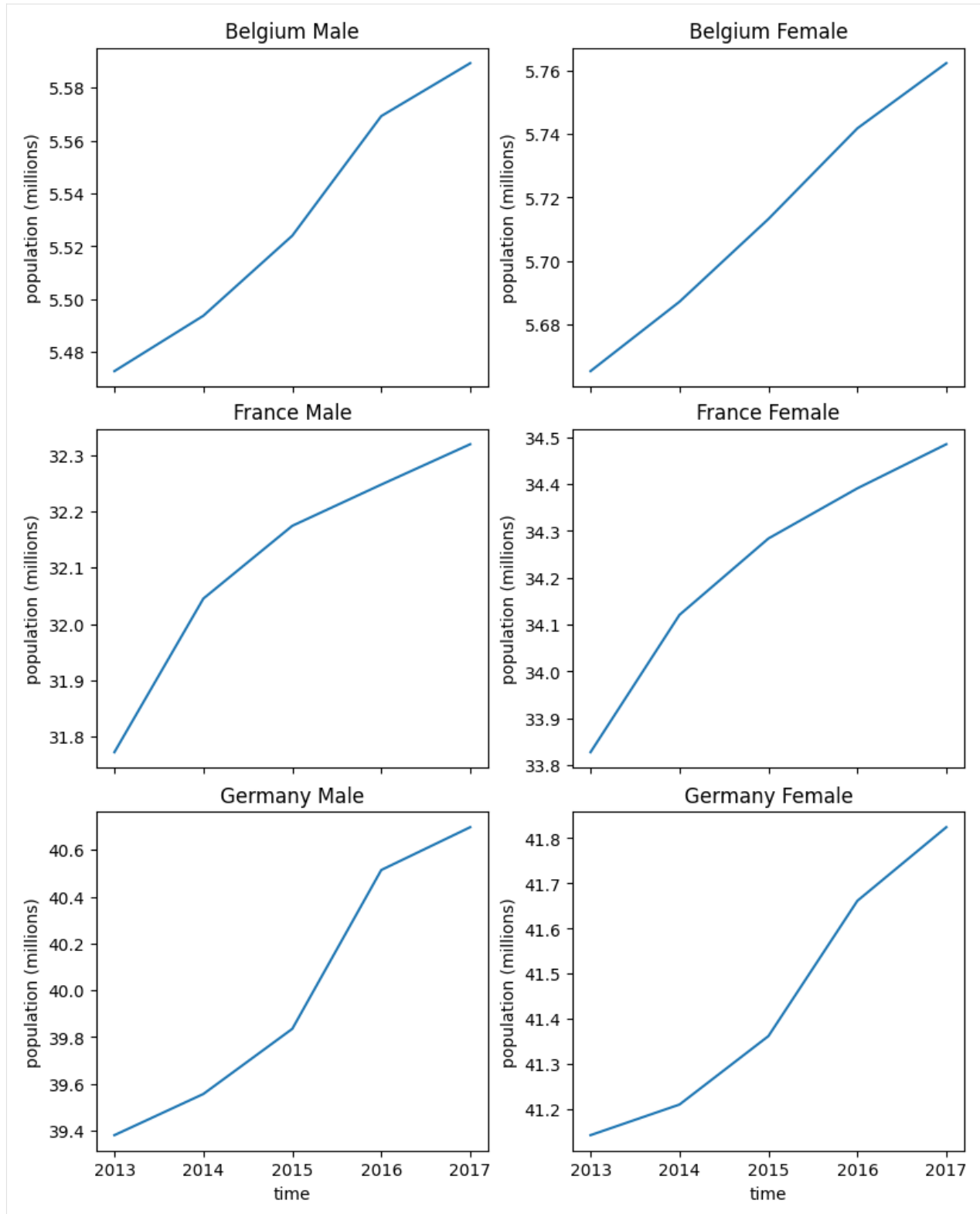
Create a *stacked* Bar plot:

```
[11]: population['Belgium'].plot.bar(title='Belgium', ylabel='population (millions)',  
    ↪ stacked=True)  
plt.show()
```



Create a multiplot figure (using subplots=axes):

```
[12]: population.plot(subplots=('country', 'gender'), sharex=True,  
                      xticks=population.time, ylabel='population (millions)',  
                      figsize=(8, 10))  
plt.show()
```



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

See [pyplot tutorial](#) for a short introduction to `matplotlib.pyplot`.

Import the LArray library:

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

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

# extract the 'population' array from the dataset
population = demography_eurostat.population
population
```

```
[2]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male  5472856  5493792  5524068  5569264  5589272
Belgium      Female 5665118  5687048  5713206  5741853  5762455
France       Male  31772665  32045129  32174258  32247386  32318973
France       Female 33827685  34120851  34283895  34391005  34485148
Germany      Male  39380976  39556923  39835457  40514123  40697118
Germany      Female 41142770  41210540  41362080  41661561  41824535
```

## 4.2.10 Inspecting Array objects

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

```
[3]: # Array summary: metadata + dimensions + description of axes
population.info
```

```
[3]: title: Population on 1 January by age and sex
source: table demo_pjan from Eurostat
3 x 2 x 5
country [3]: 'Belgium' 'France' 'Germany'
gender [2]: 'Male' 'Female'
time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes
```

Get axes

```
[4]: population.axes
```

```
[4]: AxisCollection([
    Axis(['Belgium', 'France', 'Germany'], 'country'),
    Axis(['Male', 'Female'], 'gender'),
    Axis([2013, 2014, 2015, 2016, 2017], 'time')
])
```

Get axis names

```
[5]: population.axes.names
```

```
[5]: ['country', 'gender', 'time']
```

Get number of dimensions

```
[6]: population.ndim
```

```
[6]: 3
```

Get length of each dimension

```
[7]: population.shape
```

```
[7]: (3, 2, 5)
```

Get total number of elements of the array

```
[8]: population.size
```

```
[8]: 30
```

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

```
[9]: population.dtype
```

```
[9]: dtype('int32')
```

Get size in memory

```
[10]: population.memory_used
```

```
[10]: '120 bytes'
```

## 4.2.11 Some Useful Functions

### with total

Add totals to one or several axes:

```
[11]: population.with_total('gender', label='Total')
```

```
[11]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male  5472856  5493792  5524068  5569264  5589272
      Belgium      Female 5665118  5687048  5713206  5741853  5762455
      Belgium      Total 11137974 11180840 11237274 11311117 11351727
      France      Male  31772665 32045129 32174258 32247386 32318973
      France      Female 33827685 34120851 34283895 34391005 34485148
      France      Total 65600350 66165980 66458153 66638391 66804121
      Germany      Male  39380976 39556923 39835457 40514123 40697118
      Germany      Female 41142770 41210540 41362080 41661561 41824535
      Germany      Total 80523746 80767463 81197537 82175684 82521653
```

See [with\\_total](#) for more details and examples.

### where

The where function can be used to apply some computation depending on a condition:

```
[12]: # where(condition, value if true, value if false)
      where(population < population.mean('time'), -population, population)
```

```
[12]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male -5472856 -5493792 -5524068  5569264  5589272
      Belgium      Female -5665118 -5687048 -5713206  5741853  5762455
```

(continues on next page)



(continued from previous page)

France	Male	-31772665	-32045129	32174258	32247386	32318973
France	Female	-33827685	-34120851	34283895	34391005	34485148
Germany	Male	-39380976	-39556923	-39835457	40514123	40697118
Germany	Female	-41142770	-41210540	-41362080	41661561	41824535

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

## clip

Set all data between a certain range:

```
[13]: # values below 10 millions are set to 10 millions
population.clip(minval=10**7)
```

```
[13]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male  10000000  10000000  10000000  10000000  10000000
Belgium      Female 10000000  10000000  10000000  10000000  10000000
France       Male   31772665  32045129  32174258  32247386  32318973
France       Female 33827685  34120851  34283895  34391005  34485148
Germany      Male   39380976  39556923  39835457  40514123  40697118
Germany      Female 41142770  41210540  41362080  41661561  41824535
```

```
[14]: # values above 40 millions are set to 40 millions
population.clip(maxval=4*10**7)
```

```
[14]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male   5472856  5493792  5524068  5569264  5589272
Belgium      Female 5665118  5687048  5713206  5741853  5762455
France       Male   31772665  32045129  32174258  32247386  32318973
France       Female 33827685  34120851  34283895  34391005  34485148
Germany      Male   39380976  39556923  39835457  40000000  40000000
Germany      Female 40000000  40000000  40000000  40000000  40000000
```

```
[15]: # values below 10 millions are set to 10 millions and
# values above 40 millions are set to 40 millions
population.clip(10**7, 4*10**7)
```

```
[15]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male  10000000  10000000  10000000  10000000  10000000
Belgium      Female 10000000  10000000  10000000  10000000  10000000
France       Male   31772665  32045129  32174258  32247386  32318973
France       Female 33827685  34120851  34283895  34391005  34485148
Germany      Male   39380976  39556923  39835457  40000000  40000000
Germany      Female 40000000  40000000  40000000  40000000  40000000
```

```
[16]: # Using vectors to define the lower and upper bounds
lower_bound = sequence(population.time, initial=5_500_000, inc=50_000)
upper_bound = sequence(population.time, 41_000_000, inc=100_000)

print(lower_bound, '\n')
print(upper_bound, '\n')

population.clip(lower_bound, upper_bound)
```

```
time      2013      2014      2015      2016      2017
      5500000  5550000  5600000  5650000  5700000
```

```
time      2013      2014      2015      2016      2017
      41000000  41100000  41200000  41300000  41400000
```

```
[16]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male      5500000  5550000  5600000  5650000  5700000
      Belgium      Female  5665118  5687048  5713206  5741853  5762455
      France       Male   31772665  32045129  32174258  32247386  32318973
      France       Female  33827685  34120851  34283895  34391005  34485148
      Germany      Male   39380976  39556923  39835457  40514123  40697118
      Germany      Female  41000000  41100000  41200000  41300000  41400000
```

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

## divnot0

Replace division by 0 by 0:

```
[17]: divisor = ones(population.axes, dtype=int)
      divisor['Male'] = 0
      divisor
```

```
[17]: country gender\time 2013 2014 2015 2016 2017
      Belgium      Male      0      0      0      0      0
      Belgium      Female    1      1      1      1      1
      France       Male      0      0      0      0      0
      France       Female    1      1      1      1      1
      Germany      Male      0      0      0      0      0
      Germany      Female    1      1      1      1      1
```

```
[18]: population / divisor
```

```
/tmp/ipykernel_6511/1661386825.py:1: RuntimeWarning: divide by zero encountered during
↪operation
      population / divisor
```

```
[18]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male      inf      inf      inf      inf      inf
      Belgium      Female  5665118.0  5687048.0  5713206.0  5741853.0  5762455.0
      France       Male      inf      inf      inf      inf      inf
      France       Female  33827685.0  34120851.0  34283895.0  34391005.0  34485148.0
      Germany      Male      inf      inf      inf      inf      inf
      Germany      Female  41142770.0  41210540.0  41362080.0  41661561.0  41824535.0
```

```
[19]: # we use astype(int) since the divnot0 method
      # returns a float array in this case while
      # we want an integer array
      population.divnot0(divisor).astype(int)
```

```
[19]: country gender\time      2013      2014      2015      2016      2017
      Belgium      Male      0      0      0      0      0
```

(continues on next page)

(continued from previous page)

Belgium	Female	5665118	5687048	5713206	5741853	5762455
France	Male	0	0	0	0	0
France	Female	33827685	34120851	34283895	34391005	34485148
Germany	Male	0	0	0	0	0
Germany	Female	41142770	41210540	41362080	41661561	41824535

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:

```
[20]: population.ratio('gender')
```

```
# which is equivalent to
population / population.sum('gender')
```

```
[20]: country  gender\time          2013  ...          2017
Belgium      Male      0.491369076638175  ...  0.4923719536243252
Belgium      Female    0.508630923361825  ...  0.5076280463756748
France       Male     0.48433682137366646  ...  0.48378711546852027
France       Female    0.5156631786263336  ...  0.5162128845314797
Germany      Male     0.4890604071002857  ...  0.4931689625751922
Germany      Female    0.5109395928997144  ...  0.5068310374248077
```

See *ratio* and *rationot0* for more details and examples.

## percents

```
[21]: # or, if you want the previous ratios in percents
population.percent('gender')
```

```
[21]: country  gender\time          2013  ...          2017
Belgium      Male    49.136907663817496  ...  49.237195362432516
Belgium      Female  50.863092336182504  ...  50.762804637567484
France       Male    48.433682137366645  ...  48.378711546852024
France       Female  51.566317862633355  ...  51.621288453147976
Germany      Male    48.90604071002857  ...  49.316896257519225
Germany      Female  51.09395928997143  ...  50.683103742480775
```

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.

```
[22]: # calculates 'diff[year+1] = population[year+1] - population[year]'
      population.diff('time')
```

```
[22]: country  gender\time    2014    2015    2016    2017
      Belgium      Male    20936    30276    45196    20008
      Belgium      Female  21930    26158    28647    20602
      France       Male   272464   129129    73128    71587
      France       Female  293166   163044   107110    94143
      Germany      Male   175947   278534   678666   182995
      Germany      Female   67770   151540   299481   162974
```

```
[23]: # calculates 'diff[year+2] = population[year+2] - population[year]'
      population.diff('time', d=2)
```

```
[23]: country  gender\time    2015    2016    2017
      Belgium      Male    51212    75472    65204
      Belgium      Female  48088    54805    49249
      France       Male   401593   202257   144715
      France       Female  456210   270154   201253
      Germany      Male   454481   957200   861661
      Germany      Female  219310   451021   462455
```

```
[24]: # calculates 'diff[year] = population[year+1] - population[year]'
      population.diff('time', label='lower')
```

```
[24]: country  gender\time    2013    2014    2015    2016
      Belgium      Male    20936    30276    45196    20008
      Belgium      Female  21930    26158    28647    20602
      France       Male   272464   129129    73128    71587
      France       Female  293166   163044   107110    94143
      Germany      Male   175947   278534   678666   182995
      Germany      Female   67770   151540   299481   162974
```

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]]`:

```
[25]: population.growth_rate('time')
```

```
[25]: country  gender\time    2014    ...    2017
      Belgium      Male    0.0038254249700704714    ...    0.003592575248722273
      Belgium      Female  0.0038710579373633525    ...    0.0035880403068486778
      France       Male    0.008575421671427311    ...    0.0022199318729276226
      France       Female  0.008666451753940596    ...    0.002737430906715288
      Germany      Male    0.004467817151103619    ...    0.00451681997411125
      Germany      Female  0.001647190988842025    ...    0.0039118553431063225
```

See *growth\_rate* for more details and examples.

## shift

The `shift` method drops first label of an axis and shifts all subsequent labels

```
[26]: population.shift('time')
```

```
[26]: country  gender\time      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264
Belgium      Female  5665118    5687048    5713206    5741853
France       Male    31772665   32045129   32174258   32247386
France       Female  33827685   34120851   34283895   34391005
Germany      Male    39380976   39556923   39835457   40514123
Germany      Female  41142770   41210540   41362080   41661561
```

```
[27]: # when shift is applied on an (increasing) time axis,
# it effectively brings "past" data into the future
population_shifted = population.shift('time')
stack({'population_shifted_2014': population_shifted[2014], 'population_2013':
↪population[2013]}, 'array')
```

```
[27]: country  gender\array  population_shifted_2014  population_2013
Belgium      Male          5472856          5472856
Belgium      Female        5665118          5665118
France       Male          31772665          31772665
France       Female        33827685          33827685
Germany      Male          39380976          39380976
Germany      Female        41142770          41142770
```

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*.

## 4.2.12 Working With Sessions

Import the LArray library:

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

## Three Kinds Of Sessions

They are three ways to group objects in LArray:

- *Session*: is an ordered dict-like container with special I/O methods. Although the *autocomplete\** feature on the objects stored in the session is available in the larray-editor, it is not available in development tools like PyCharm making it cumbersome to use.
- *CheckedSession*: provides the same methods as Session objects but are defined in a completely different way (see example below). The *autocomplete\** feature is both available in the larray-editor and in development tools (PyCharm). In addition, the type of each stored object is protected. Optionally, it is possible to constrain the axes and dtype of arrays using *CheckedArray*.
- *CheckedParameters*: is a special version of *CheckedSession* in which the value of all stored objects (parameters) is frozen after initialization.

\* *Autocomplete* is the feature in which development tools try to predict the variable or function a user intends to enter after only a few characters have been typed (like word completion in cell phones).

## Creating Sessions

### Session

Create a session:

```
[3]: # define some scalars, axes and arrays
variant = 'baseline'

country = Axis('country=Belgium,France,Germany')
gender = Axis('gender=Male,Female')
time = Axis('time=2013..2017')

population = zeros([country, gender, time])
births = zeros([country, gender, time])
deaths = zeros([country, gender, time])

[4]: # create an empty session and objects one by one after
s = Session()
s.variant = variant
s.country = country
s.gender = gender
s.time = time
s.population = population
s.births = births
s.deaths = deaths

print(s.summary())

variant: baseline
country: country ['Belgium' 'France' 'Germany'] (3)
gender: gender ['Male' 'Female'] (2)
time: time [2013 2014 2015 2016 2017] (5)
population: country, gender, time (3 x 2 x 5) [float64]
births: country, gender, time (3 x 2 x 5) [float64]
deaths: country, gender, time (3 x 2 x 5) [float64]
```

```
[5]: # or create a session in one step by passing all objects to the constructor
s = Session(variant=variant, country=country, gender=gender, time=time,
            population=population, births=births, deaths=deaths)
```

```
print(s.summary())
```

```
variant: baseline
country: country ['Belgium' 'France' 'Germany'] (3)
gender: gender ['Male' 'Female'] (2)
time: time [2013 2014 2015 2016 2017] (5)
population: country, gender, time (3 x 2 x 5) [float64]
births: country, gender, time (3 x 2 x 5) [float64]
deaths: country, gender, time (3 x 2 x 5) [float64]
```

## CheckedSession

The syntax to define a checked-session is a bit specific:

```
class MySession(CheckedSession):
    # Variables can be declared in two ways:
    # a) by specifying only the type of the variable (to be initialized later)
    var1: Type
    # b) by giving an initialization value.
    # In that case, the type is deduced from the initialization value
    var2 = initialization value
    # Additionally, axes and dtype of Array variables can be constrained
    # using the special type CheckedArray
    arr1: CheckedArray([list, of, axes], dtype) = initialization value
```

Check the example below:

```
[6]: class Demography(CheckedSession):
    # (convention is to declare parameters (read-only objects) in capital letters)
    # Declare 'VARIANT' parameter as of type string.
    # 'VARIANT' will be initialized when a 'Demography' session will be created
    VARIANT: str
    # declare variables with an initialization value.
    # Their type is deduced from their initialization value.
    COUNTRY = Axis('country=Belgium,France,Germany')
    GENDER = Axis('gender=Male,Female')
    TIME = Axis('time=2013..2017')
    population = zeros([COUNTRY, GENDER, TIME], dtype=int)
    births = zeros([COUNTRY, GENDER, TIME], dtype=int)
    # declare 'deaths' with constrained axes and dtype.
    # Its type (Array), axes and dtype are not modifiable.
    # It will be initialized with 0
    deaths: CheckedArray([COUNTRY, GENDER, TIME], int) = 0

d = Demography(VARIANT='baseline')

print(d.summary())
```

```
VARIANT: baseline
deaths: country, gender, time (3 x 2 x 5) [int64]
COUNTRY: country ['Belgium' 'France' 'Germany'] (3)
GENDER: gender ['Male' 'Female'] (2)
TIME: time [2013 2014 2015 2016 2017] (5)
population: country, gender, time (3 x 2 x 5) [int64]
births: country, gender, time (3 x 2 x 5) [int64]
```

## 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:

```
[7]: # 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')
s = 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')
s = 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')
s = Session(filepath_hdf)

print(s.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
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]
population: country, gender, time (3 x 2 x 5) [int32]
population_5_countries: country, gender, time (5 x 2 x 5) [int32]
population_benelux: country, gender, time (3 x 2 x 5) [int32]
citizenship: citizenship ['Belgium' 'Luxembourg' 'Netherlands'] (3)
country: country ['Belgium' 'France' 'Germany'] (3)
country_benelux: country ['Belgium' 'Luxembourg' 'Netherlands'] (3)
gender: gender ['Male' 'Female'] (2)
time: time [2013 2014 2015 2016 2017] (5)
```

(continues on next page)



(continued from previous page)

```
even_years: time[2014 2016] >> even_years (2)
odd_years: time[2013 2015 2017] >> odd_years (3)
```

- 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:

```
[8]: # create a session containing 3 axes, 2 groups and one array 'population'
      filepath = get_example_filepath('population_only.xlsx')
      s = Session(filepath)

      print(s.summary())

population: country, gender, time (3 x 2 x 3) [int64]
```

```
[9]: # 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')
      s.load(filepath)

      print(s.summary())

population: country, gender, time (3 x 2 x 3) [int64]
births: country, gender, time (3 x 2 x 3) [int64]
deaths: country, gender, time (3 x 2 x 3) [int64]
```

The load method offers some options:

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

```
[10]: births_and_deaths_session = Session()

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

      print(births_and_deaths_session.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
births: country, gender, time (3 x 2 x 5) [int32]
deaths: country, gender, time (3 x 2 x 5) [int32]
```

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

```
[11]: s = Session()

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

opening /home/docs/checkouts/readthedocs.org/user_builds/larray/envs/stable/lib/python3.
↪ 11/site-packages/larray/tests/data/demography_eurostat.h5
loading Array object births ... done
loading Array object deaths ... done
loading Array object immigration ... done
```

(continues on next page)

(continued from previous page)

```
loading Array object population ... done
loading Array object population_5_countries ... done
loading Array object population_benelux ... done
loading Axis_Backward_Comp object citizenship ... done
loading Axis_Backward_Comp object country ... done
loading Axis_Backward_Comp object country_benelux ... done
loading Axis_Backward_Comp object gender ... done
loading Axis_Backward_Comp object time ... done
loading Group_Backward_Comp object even_years ... done
loading Group_Backward_Comp object odd_years ... done
```

## 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:

```
[12]: # save items of a session in CSV files.
      # Here, the save method will create a 'demography' directory in which CSV files will be_
      ↪written
      s.save('demography')

      # save the session to an HDF5 file
      s.save('demography.h5')

      # save the session to an Excel file
      s.save('demography.xlsx')
```

---

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:

```
[13]: # use the names argument to only save births and deaths arrays
      s.save('demography.h5', names=['births', 'deaths'])

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

[13]: ['births', 'deaths']
```

- 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:

```
[14]: population = read_csv('./demography/population.csv')
      pop_ses = Session([('population', population)])

      # 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_
```

(continues on next page)

(continued from previous page)

```
↪overwritten.
pop_ses.save('demography.h5', overwrite=False)

# load session saved in 'demography.h5' to see its content
Session('demography.h5').names
```

```
[14]: ['births', 'deaths', 'population']
```

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

```
[15]: # with display=True, the save method will print a message
# each time an item is dumped
s.save('demography.h5', display=True)
```

```
dumping births ... done
dumping deaths ... done
dumping immigration ... done
dumping population ... done
dumping population_5_countries ... done
dumping population_benelux ... done
dumping citizenship ... done
dumping country ... done
dumping country_benelux ... done
dumping gender ... done
dumping time ... done
dumping even_years ... done
dumping odd_years ... done
```

## 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!):

```
[16]: # load a session representing the results of a demographic model
filepath_hdf = get_example_filepath('demography_eurostat.h5')
s = Session(filepath_hdf)

# print the content of the session
print(s.names)

['births', 'citizenship', 'country', 'country_benelux', 'deaths', 'even_years', 'gender',
↪ 'immigration', 'odd_years', 'population', 'population_5_countries', 'population_
↪benelux', 'time']
```

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:

```
[17]: # print the content of the session
print(s.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
```

(continues on next page)

(continued from previous page)

```

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]
population: country, gender, time (3 x 2 x 5) [int32]
population_5_countries: country, gender, time (5 x 2 x 5) [int32]
population_benelux: country, gender, time (3 x 2 x 5) [int32]
citizenship: citizenship ['Belgium' 'Luxembourg' 'Netherlands'] (3)
country: country ['Belgium' 'France' 'Germany'] (3)
country_benelux: country ['Belgium' 'Luxembourg' 'Netherlands'] (3)
gender: gender ['Male' 'Female'] (2)
time: time [2013 2014 2015 2016 2017] (5)
even_years: time[2014 2016] >> even_years (2)
odd_years: time[2013 2015 2017] >> odd_years (3)

```

## Selecting And Filtering Items

Session objects work like ordinary dict Python objects. To select an item, use the usual syntax `<session_var>['<item_name>']`:

```
[18]: s['population']
```

```

[18]: country  gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535

```

A simpler way consists in the use the syntax `<session_var>.<item_name>`:

```
[19]: s.population
```

```

[19]: country  gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535

```

**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]`:

```
[20]: s_selected = s['population', 'births', 'deaths']
```

```
s_selected.names
```

```
[20]: ['births', 'deaths', 'population']
```

**Warning:** The same selection as above can be applied on a checked-session **but the returned object is a normal session and NOT a checked-session**. This means that you will loose all the benefits (autocomplete, protection on type, axes and dtype) of checked-sessions.

```
[21]: d_selected = d['births', 'deaths']

# test if v_selected is a checked-session
print('is still a check-session?', isinstance(d_selected, CheckedSession))
#test if v_selected is a normal session
print('is now a normal session?', isinstance(d_selected, Session))

is still a check-session? False
is now a normal session? True
```

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:

```
[22]: # select only arrays of a session
s.filter(kind=Array)

[22]: Session(births, deaths, immigration, population, population_5_countries, population_
↪benelux)
```

```
[23]: # selection all items with a name starting with a letter between a and k
s.filter(pattern='[a-k]*')

[23]: Session(births, deaths, immigration, citizenship, country, country_benelux, gender, even_
↪years)
```

**Warning:** Using the *filter()* method on a checked-session **will return a normal session and NOT a checked-session**. This means that you will loose all the benefits (autocomplete, protection on type, axes and dtype) of checked-sessions.

```
[24]: d_filtered = d.filter(pattern='[a-k]*')

# test if v_selected is a checked-session
print('is still a check-session?', isinstance(d_filtered, CheckedSession))
#test if v_selected is a normal session
print('is now a normal session?', isinstance(d_filtered, Session))

is still a check-session? False
is now a normal session? True
```

## Iterating over Items

Like the built-in Python dict objects, Session objects provide methods to iterate over items:

```
[25]: # iterate over item names
      for key in s.keys():
          print(key)
```

```
births
deaths
immigration
population
population_5_countries
population_benelux
citizenship
country
country_benelux
gender
time
even_years
odd_years
```

```
[26]: # iterate over items
      for value in s.values():
          if isinstance(value, Array):
              print(value.info)
          else:
              print(repr(value))
          print()
```

```
title: Live births by mother's age and newborn's sex
source: table demo_fasec from Eurostat
3 x 2 x 5
  country [3]: 'Belgium' 'France' 'Germany'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes
```

```
title: Deaths by age and sex
source: table demo_magec from Eurostat
3 x 2 x 5
  country [3]: 'Belgium' 'France' 'Germany'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes
```

```
title: Immigration by age group, sex and citizenship
source: table migr_imm1ctz from Eurostat
3 x 3 x 2 x 5
  country [3]: 'Belgium' 'Luxembourg' 'Netherlands'
  citizenship [3]: 'Belgium' 'Luxembourg' 'Netherlands'
  gender [2]: 'Male' 'Female'
```

(continues on next page)

(continued from previous page)

```

time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 360 bytes

title: Population on 1 January by age and sex
source: table demo_pjan from Eurostat
3 x 2 x 5
country [3]: 'Belgium' 'France' 'Germany'
gender [2]: 'Male' 'Female'
time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes

title: Population on 1 January by age and sex (Benelux + France + Germany)
source: table demo_pjan from Eurostat
5 x 2 x 5
country [5]: 'Belgium' 'France' 'Germany' 'Luxembourg' 'Netherlands'
gender [2]: 'Male' 'Female'
time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 200 bytes

title: Population on 1 January by age and sex (Benelux)
source: table demo_pjan from Eurostat
3 x 2 x 5
country [3]: 'Belgium' 'Luxembourg' 'Netherlands'
gender [2]: 'Male' 'Female'
time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes

Axis(['Belgium', 'Luxembourg', 'Netherlands'], 'citizenship')

Axis(['Belgium', 'France', 'Germany'], 'country')

Axis(['Belgium', 'Luxembourg', 'Netherlands'], 'country')

Axis(['Male', 'Female'], 'gender')

Axis([2013, 2014, 2015, 2016, 2017], 'time')

time[2014, 2016] >> 'even_years'

time[2013, 2015, 2017] >> 'odd_years'

```

```

[27]: # iterate over names and items
      for key, value in s.items():
          if isinstance(value, Array):
              print(key, ':')
              print(value.info)
          else:

```

(continues on next page)

(continued from previous page)

```

    print(key, ':', repr(value))
print()

births :
title: Live births by mother's age and newborn's sex
source: table demo_fasec from Eurostat
3 x 2 x 5
  country [3]: 'Belgium' 'France' 'Germany'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes

deaths :
title: Deaths by age and sex
source: table demo_magec from Eurostat
3 x 2 x 5
  country [3]: 'Belgium' 'France' 'Germany'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes

immigration :
title: Immigration by age group, sex and citizenship
source: table migr_imm1ctz from Eurostat
3 x 3 x 2 x 5
  country [3]: 'Belgium' 'Luxembourg' 'Netherlands'
  citizenship [3]: 'Belgium' 'Luxembourg' 'Netherlands'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 360 bytes

population :
title: Population on 1 January by age and sex
source: table demo_pjan from Eurostat
3 x 2 x 5
  country [3]: 'Belgium' 'France' 'Germany'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes

population_5_countries :
title: Population on 1 January by age and sex (Benelux + France + Germany)
source: table demo_pjan from Eurostat
5 x 2 x 5
  country [5]: 'Belgium' 'France' 'Germany' 'Luxembourg' 'Netherlands'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32

```

(continues on next page)



(continued from previous page)

```

memory used: 200 bytes

population_benelux :
title: Population on 1 January by age and sex (Benelux)
source: table demo_pjan from Eurostat
3 x 2 x 5
  country [3]: 'Belgium' 'Luxembourg' 'Netherlands'
  gender [2]: 'Male' 'Female'
  time [5]: 2013 2014 2015 2016 2017
dtype: int32
memory used: 120 bytes

citizenship : Axis(['Belgium', 'Luxembourg', 'Netherlands'], 'citizenship')

country : Axis(['Belgium', 'France', 'Germany'], 'country')

country_benelux : Axis(['Belgium', 'Luxembourg', 'Netherlands'], 'country')

gender : Axis(['Male', 'Female'], 'gender')

time : Axis([2013, 2014, 2015, 2016, 2017], 'time')

even_years : time[2014, 2016] >> 'even_years'

odd_years : time[2013, 2015, 2017] >> 'odd_years'

```

## Manipulating Checked Sessions

**Note:** this section only concerns objects declared in checked-sessions.

Let's create a simplified version of the *Demography* checked-session we have defined above:

```

[28]: class Demography(CheckedSession):
        COUNTRY = Axis('country=Belgium,France,Germany')
        GENDER = Axis('gender=Male,Female')
        TIME = Axis('time=2013..2017')
        population = zeros([COUNTRY, GENDER, TIME], dtype=int)
        # declare the deaths array with constrained axes and dtype
        deaths: CheckedArray([COUNTRY, GENDER, TIME], int) = 0

d = Demography()

print(d.summary())

deaths: country, gender, time (3 x 2 x 5) [int64]
COUNTRY: country ['Belgium' 'France' 'Germany'] (3)
GENDER: gender ['Male' 'Female'] (2)
TIME: time [2013 2014 2015 2016 2017] (5)
population: country, gender, time (3 x 2 x 5) [int64]

```

One of the specificities of checked-sessions is that the type of the contained objects is protected (it cannot change). Any attempt to assign a value of different type will raise an error:

```
[29]: # The population variable was initialized with the zeros() function which returns an
      ↪ Array object.
      # The declared type of the population variable is Array and is protected
      d.population = Axis('population=child,teenager,adult,elderly')

ArbitraryTypeError: instance of Array expected
```

The *death* array has been declared as a *CheckedArray*. As a consequence, its axes are protected. Trying to assign a value with incompatible axes raises an error:

```
[30]: AGE = Axis('age=0..100')
      d.deaths = zeros([d.COUNTRY, AGE, d.GENDER, d.TIME])

ValueError: Array 'deaths' was declared with axes {country, gender, time} but got array
      ↪ with axes {country, age, gender, time} (unexpected {age} axis)
```

The *deaths* array is also constrained by its declared dtype *int*. This means that if you try to assign a value of type *float* instead of *int*, the value will be converted to *int* if possible:

```
[31]: d.deaths = 1.2
      d.deaths
```

country	gender\time	2013	2014	2015	2016	2017
Belgium	Male	1	1	1	1	1
Belgium	Female	1	1	1	1	1
France	Male	1	1	1	1	1
France	Female	1	1	1	1	1
Germany	Male	1	1	1	1	1
Germany	Female	1	1	1	1	1

or raise an error:

```
[32]: d.deaths = 'undead'

ValueError: invalid literal for int() with base 10: 'undead'
```

It is possible to add a new variable after the checked-session has been initialized but in that case, a warning message is printed (in case you misspelled the name of variable while trying to modify it):

```
[33]: # misspell population (forgot the 'a')
      d.popultion = 0

/tmp/ipykernel_7030/1566890367.py:2: UserWarning: 'popultion' is not declared in
      ↪ 'Demography'
      d.popultion = 0
```

## Arithmetic Operations On Sessions

Session objects accept binary operations with a scalar:

```
[34]: # get population, births and deaths in millions
s_div = s / 1e6
```

```
s_div.population
```

```
[34]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male    5.472856    5.493792    5.524068    5.569264    5.589272
Belgium      Female  5.665118    5.687048    5.713206    5.741853    5.762455
France       Male   31.772665   32.045129   32.174258   32.247386   32.318973
France       Female  33.827685   34.120851   34.283895   34.391005   34.485148
Germany      Male   39.380976   39.556923   39.835457   40.514123   40.697118
Germany      Female  41.14277    41.21054    41.36208    41.661561   41.824535
```

with an array (please read the documentation of the [random.choice](#) function first if you don't know it):

```
[35]: from larray import random
random_increment = random.choice([-1, 0, 1], p=[0.3, 0.4, 0.3], axes=s.population.axes)
↳ * 1000
random_increment
```

```
[35]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male         0      1000     -1000      1000      1000
Belgium      Female    1000      1000     -1000         0     -1000
France       Male         0      1000     -1000      1000      1000
France       Female   -1000         0      1000      1000      1000
Germany      Male         0      1000         0     -1000         0
Germany      Female    1000      1000     -1000         0      1000
```

```
[36]: # add some variables of a session by a common array
s_rand = s['population', 'births', 'deaths'] + random_increment
```

```
s_rand.population
```

```
[36]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5494792    5523068    5570264    5590272
Belgium      Female  5666118    5688048    5712206    5741853    5761455
France       Male   31772665   32046129   32173258   32248386   32319973
France       Female  33826685   34120851   34284895   34392005   34486148
Germany      Male   39380976   39557923   39835457   40513123   40697118
Germany      Female  41143770   41211540   41361080   41661561   41825535
```

with another session:

```
[37]: # compute the difference between each array of the two sessions
s_diff = s - s_rand
```

```
s_diff.births
```

```
[37]: country gender\time      2013      2014      2015      2016      2017
Belgium      Male         0     -1000      1000     -1000     -1000
Belgium      Female   -1000     -1000      1000         0      1000
```

(continues on next page)

(continued from previous page)

France	Male	0	-1000	1000	-1000	-1000
France	Female	1000	0	-1000	-1000	-1000
Germany	Male	0	-1000	0	1000	0
Germany	Female	-1000	-1000	1000	0	-1000

## 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:

```
[38]: # 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_with_next_year = s.apply(add_next_year)
```

```
print('population array before calling apply:')
print(s.population)
print()
print('population array after calling apply:')
print(s_with_next_year.population)
```

```
population array before calling apply:
country gender\time      2013      2014      2015      2016      2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535
```

```
population array after calling apply:
country gender\time      2013      2014      2015      2016      2017      2018
Belgium      Male    5472856    5493792    5524068    5569264    5589272      0
Belgium      Female  5665118    5687048    5713206    5741853    5762455      0
France       Male    31772665   32045129   32174258   32247386   32318973      0
France       Female  33827685   34120851   34283895   34391005   34485148      0
Germany      Male    39380976   39556923   39835457   40514123   40697118      0
Germany      Female  41142770   41210540   41362080   41661561   41824535      0
```

It is possible to pass a function with additional arguments:

```
[39]: # add the next year to all arrays.
# Use the 'copy_values_from_last_year flag' to indicate
# whether 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]
```

(continues on next page)

(continued from previous page)

```

        value = array[last_year] if copy_values_from_last_year else 0
        return array.append('time', value, last_year + 1)
    else:
        return array

s_with_next_year = s.apply(add_next_year, True)

print('population array before calling apply:')
print(s.population)
print()
print('population array after calling apply:')
print(s_with_next_year.population)

```

```

population array before calling apply:
country  gender\time      2013      2014      2015      2016      2017
Belgium   Male   5472856   5493792   5524068   5569264   5589272
Belgium   Female 5665118   5687048   5713206   5741853   5762455
France    Male   31772665  32045129  32174258  32247386  32318973
France    Female 33827685  34120851  34283895  34391005  34485148
Germany   Male   39380976  39556923  39835457  40514123  40697118
Germany   Female 41142770  41210540  41362080  41661561  41824535

```

```

population array after calling apply:
country  gender\time      2013      2014      2015      2016      2017      2018
Belgium   Male   5472856   5493792   5524068   5569264   5589272   5589272
Belgium   Female 5665118   5687048   5713206   5741853   5762455   5762455
France    Male   31772665  32045129  32174258  32247386  32318973  32318973
France    Female 33827685  34120851  34283895  34391005  34485148  34485148
Germany   Male   39380976  39556923  39835457  40514123  40697118  40697118
Germany   Female 41142770  41210540  41362080  41661561  41824535  41824535

```

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.

```

[40]: # load a session representing the results of a demographic model
filepath_hdf = get_example_filepath('demography_eurostat.h5')
s = Session(filepath_hdf)

# create a copy of the original session
s_copy = s.copy()

```

```
[41]: # 'element_equals' compare arrays one by one
s.element_equals(s_copy)
```

```
[41]: name  births  deaths  ...  time  even_years  odd_years
      True    True   ...   True         True         True
```

```
[42]: # 'equals' returns True if all items of the two sessions have exactly the same items
s.equals(s_copy)
```

```
[42]: True
```

```
[43]: # slightly modify the 'population' array for some labels combination
s_copy.population += random_increment
```

```
[44]: # the 'population' array is different between the two sessions
s.element_equals(s_copy)
```

```
[44]: name  births  deaths  ...  time  even_years  odd_years
      True    True   ...   True         True         True
```

```
[45]: # 'equals' returns False if at least one item of the two sessions are different in values,
      ↪ or axes
s.equals(s_copy)
```

```
[45]: False
```

```
[46]: # reset the 'copy' session as a copy of the original session
s_copy = s.copy()
```

```
      # add an array to the 'copy' session
s_copy.gender_ratio = s_copy.population.ratio('gender')
```

```
[47]: # the 'gender_ratio' array is not present in the original session
s.element_equals(s_copy)
```

```
[47]: name  births  deaths  ...  even_years  odd_years  gender_ratio
      True    True   ...         True         True         False
```

```
[48]: # 'equals' returns False if at least one item is not present in the two sessions
s.equals(s_copy)
```

```
[48]: False
```

The == operator return a new session with boolean arrays with elements compared element-wise:

```
[49]: # reset the 'copy' session as a copy of the original session
s_copy = s.copy()
```

```
      # slightly modify the 'population' array for some labels combination
s_copy.population += random_increment
```

```
[50]: s_check_same_values = s == s_copy
```

```
s_check_same_values.population
```

```
[50]: country gender\time 2013 2014 2015 2016 2017
      Belgium      Male   True  False  False  False  False
      Belgium      Female False  False  False   True  False
      France       Male   True  False  False  False  False
      France       Female False   True  False  False  False
      Germany      Male   True  False   True  False   True
      Germany      Female False  False  False   True  False
```

This also works for axes and groups:

```
[51]: s_check_same_values.time
```

```
[51]: time 2013 2014 2015 2016 2017
      True  True  True  True  True
```

The `!=` operator does the opposite of `==` operator:

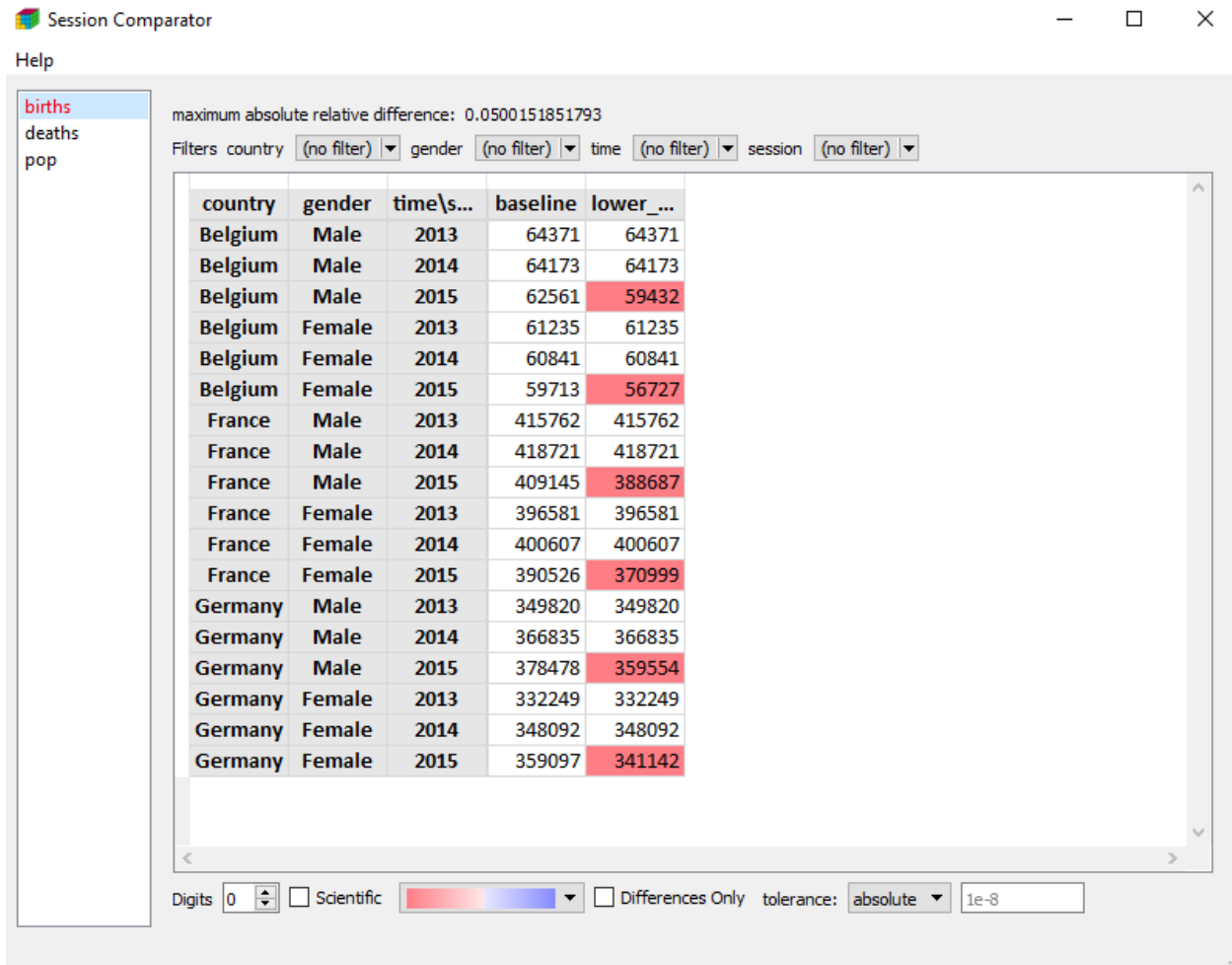
```
[52]: s_check_different_values = s != s_copy
```

```
s_check_different_values.population
```

```
[52]: country gender\time 2013 2014 2015 2016 2017
      Belgium      Male  False   True   True   True   True
      Belgium      Female  True   True   True  False   True
      France       Male  False   True   True   True   True
      France       Female  True  False   True   True   True
      Germany      Male  False   True  False   True  False
      Germany      Female  True   True   True  False   True
```

A more visual way is to use the `compare` function which will open the Editor.

```
compare(s, s_alternative, names=['baseline', 'lower_birth_rate'])
```



### 4.2.13 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]: population = load_example_data('demography_eurostat').population
population
```

```
[2]: country  gender\time    2013    2014    2015    2016    2017
Belgium      Male   5472856   5493792   5524068   5569264   5589272
Belgium      Female 5665118   5687048   5713206   5741853   5762455
France       Male   31772665  32045129  32174258  32247386  32318973
France       Female 33827685  34120851  34283895  34391005  34485148
Germany      Male   39380976  39556923  39835457  40514123  40697118
Germany      Female 41142770  41210540  41362080  41661561  41824535
```

```
[3]: df = population.to_frame()
df
```



```
[3]: time          2013      2014      2015      2016      2017
country gender
Belgium Male    5472856    5493792    5524068    5569264    5589272
        Female  5665118    5687048    5713206    5741853    5762455
France  Male    31772665   32045129   32174258   32247386   32318973
        Female  33827685   34120851   34283895   34391005   34485148
Germany Male    39380976   39556923   39835457   40514123   40697118
        Female  41142770   41210540   41362080   41661561   41824535
```

Inversely, to convert a DataFrame into an Array object, use the function `asarray()`:

```
[4]: population = asarray(df)
population
```

```
[4]: country gender\time    2013    2014    2015    2016    2017
Belgium      Male    5472856    5493792    5524068    5569264    5589272
Belgium      Female  5665118    5687048    5713206    5741853    5762455
France       Male    31772665   32045129   32174258   32247386   32318973
France       Female  33827685   34120851   34283895   34391005   34485148
Germany      Male    39380976   39556923   39835457   40514123   40697118
Germany      Female  41142770   41210540   41362080   41661561   41824535
```

## 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__`**(*labels*, *name=None*)

## Methods

<code>__init__(labels[, name])</code>	
<code>align(other[, join])</code>	Align axis with other object using specified join method.
<code>all([name])</code>	(Deprecated) Return a group containing all labels.
<code>apply(func)</code>	Return a new axis with the labels transformed by func.
<code>astype(dtype[, casting])</code>	Cast labels to a specified type.
<code>by(length[, step, template])</code>	Split axis into several groups of specified length.
<code>containing(substring)</code>	Return a group with all the labels containing the specified substring.
<code>copy()</code>	Return a copy of the axis.
<code>difference(other)</code>	Return axis with the (set) difference of this axis labels and other labels.
<code>endingwith(suffix)</code>	Return a group with the labels ending with the specified string.
<code>endswith(**kwargs)</code>	
<code>equals(other)</code>	Check if this axis is equal to another axis.
<code>extend(labels)</code>	Append new labels to an axis or increase its length in case of wildcard axis.
<code>group(*args[, name])</code>	
<code>ignore_labels()</code>	Return a wildcard axis with the same name and length than this axis.
<code>index(key)</code>	Translate a label key to its numerical index counterpart.
<code>insert(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(other)</code>	Return axis with the (set) intersection of this axis labels and other labels.
<code>iscompatible(other)</code>	Check if this axis is compatible with another axis.
<code>labels_summary()</code>	Return a short representation of the labels.
<code>matches(**kwargs)</code>	
<code>matching([deprecated, pattern, regex])</code>	Return a group with all the labels matching the specified pattern or regular expression.
<code>max()</code>	Get maximum of labels.
<code>min()</code>	Get minimum of labels.
<code>rename(name)</code>	Rename the axis.
<code>replace(old[, new])</code>	Return a new axis with some labels replaced.
<code>split([sep, names, regex, return_labels])</code>	Split axis and returns a list of Axis.
<code>startingwith(prefix)</code>	Return a group with the labels starting with the specified string.
<code>startswith(**kwargs)</code>	
<code>subaxis(key)</code>	Return an axis for a sub-array.
<code>to_hdf(filepath[, key])</code>	Write axis to a HDF file.

continues on next page

Table 1 – continued from previous page

<code>translate(**kwargs)</code>	
<code>union(other)</code>	Return axis with the union of this axis labels and other labels.

Attributes

name	
dtype	
<code>i</code>	Take a subset using positions along the axis instead of labels.
id	
iswildcard	
labels	Labels of the axis.

Exploring

Axis.name	Name of the axis. None in the case of an anonymous axis.
Axis.labels	Labels of the axis.
Axis.labels_summary	Short representation of the labels.
Axis.dtype	Data type for the axis labels.

Copying

<code>Axis.copy()</code>	Return a copy of the axis.
--------------------------	----------------------------

`larray.Axis.copy`

`Axis.copy()` → *Axis*  
Return a copy of the axis.

## Searching

<code>Axis.index(key)</code>	Translate a label key to its numerical index counterpart.
<code>Axis.containing(substring)</code>	Return a group with all the labels containing the specified substring.
<code>Axis.startingwith(prefix)</code>	Return a group with the labels starting with the specified string.
<code>Axis.endingwith(suffix)</code>	Return a group with the labels ending with the specified string.
<code>Axis.matching([deprecated, pattern, regex])</code>	Return a group with all the labels matching the specified pattern or regular expression.
<code>Axis.min()</code>	Get minimum of labels.
<code>Axis.max()</code>	Get maximum of labels.

## larray.Axis.index

`Axis.index(key) → Union[int, ndarray, slice]`

Translate a label key to its numerical index counterpart.

### Parameters

#### key

[key] Everything usable as a key.

### Returns

#### int, slice, np.ndarray or Array

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**(*substring*) → *LGroup*

Return 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**(*prefix*) → *LGroup*

Return 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**(*suffix*) → *LGroup*

Return 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**(*deprecated=None, pattern=None, regex=None*) → *LGroup*

Return 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"
```

(continues on next page)

(continued from previous page)

```
>>> 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']
```

## larray.Axis.min

`Axis.min()` → Union[bool, int, float, str, bytes, generic]

Get minimum of labels.

### Returns

#### label

Label with minimum value.

**Warning:** Fails on non-numeric labels.

## Examples

```
>>> time = Axis('time=1991..2020')
>>> time.min()
1991
```

## larray.Axis.max

`Axis.max()` → Union[bool, int, float, str, bytes, generic]

Get maximum of labels.

### Returns

#### label

Label with maximum value.

**Warning:** Fails on non-numeric labels.



## Examples

```
>>> time = Axis('time=1991..2020')
>>> time.max()
2020
```

## Modifying/Selecting

<code>Axis.__getitem__(key)</code>	Return a group (list or unique element) of label(s) usable in <code>.sum</code> or <code>.filter</code> .
<code>Axis.i</code>	Take a subset using positions along the axis instead of labels.
<code>Axis.by(length[, step, template])</code>	Split axis into several groups of specified length.
<code>Axis.rename(name)</code>	Rename the axis.
<code>Axis.extend(labels)</code>	Append new labels to an axis or increase its length in case of wildcard axis.
<code>Axis.insert(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(old[, new])</code>	Return a new axis with some labels replaced.
<code>Axis.apply(func)</code>	Return a new axis with the labels transformed by <i>func</i> .
<code>Axis.union(other)</code>	Return axis with the union of this axis labels and other labels.
<code>Axis.intersection(other)</code>	Return axis with the (set) intersection of this axis labels and other labels.
<code>Axis.difference(other)</code>	Return axis with the (set) difference of this axis labels and other labels.
<code>Axis.align(other[, join])</code>	Align axis with other object using specified join method.
<code>Axis.split([sep, names, regex, return_labels])</code>	Split axis and returns a list of Axis.
<code>Axis.ignore_labels()</code>	Return a wildcard axis with the same name and length than this axis.
<code>Axis.astype(dtype[, casting])</code>	Cast labels to a specified type.

## larray.Axis.\_\_getitem\_\_

`Axis.__getitem__(key) → Union[LGroup, List[Group], Tuple[Group, ...]]`

Return 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

Take 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**(length, step=None, template=None) → **Tuple**[Group, ...]

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**(*name*) → *Axis*

Rename 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.extend

**Axis.extend**(*labels*) → *Axis*

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 this axis is (not) wildcard
```

**larray.Axis.insert**

**Axis.insert**(*new\_labels*, *before=None*, *after=None*) → *Axis*

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')
```

(continues on next page)

(continued from previous page)

```

>>> 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**(*old*, *new=None*) → *Axis*

Return 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**(*func*) → *Axis*

Return 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**(*other*) → *Axis*

Return 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**(*other*) → *Axis*

Return 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')
Axis(['a1', 'a2'], 'a')
>>> a.intersection(['a1', 'a2', 'a3'])
Axis(['a1', 'a2'], 'a')
```

### larray.Axis.difference

Axis.**difference**(*other*) → *Axis*

Return 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(other, join='outer') → Axis`

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(sep='_', names=None, regex=None, return_labels=False) → Union[List[Axis], Tuple[List[Axis], Tuple[Group]]]`

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.

**return\_labels**

[bool, optional] Whether 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')]
>>> ab = Axis('ab=a0_b0,a0_b1')
>>> ab.split(names=["a", "b"])
[Axis(['a0'], 'a'), Axis(['b0', 'b1'], 'b')]
```

**larray.Axis.ignore\_labels**

`Axis.ignore_labels()` → *Axis*

Return 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')
```

**larray.Axis.astype**

`Axis.astype(dtype: Union[str, dtype], casting: str = 'unsafe')` → *Axis*

Cast labels to a specified type.

**Parameters****dtype: str or dtype**

Typecode or data-type to which the labels are cast.

**casting: str, optional**

Controls what kind of data casting may occur. Defaults to *unsafe*.

- *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.

### Returns

#### Axis

Axis with labels converted to the new type.

### Examples

```
>>> from larray import ndtest
>>> arr = ndtest('time=2015..2020')
>>> arr = arr.with_total()
>>> arr
time  2015  2016  2017  2018  2019  2020  total
      0     1     2     3     4     5     15
>>> arr = arr.drop('total')
>>> time = arr.time
>>> time
Axis([2015, 2016, 2017, 2018, 2019, 2020], 'time')
>>> time.dtype
dtype('O')
>>> time = time.astype(int)
>>> time.dtype
dtype('int64')
```

### Testing

<code>Axis.iscompatible(other)</code>	Check if this axis is compatible with another axis.
<code>Axis.equals(other)</code>	Check if this axis is equal to another axis.

### `larray.Axis.iscompatible`

`Axis.iscompatible(other) → bool`

Check if this axis is compatible with another axis.

- two axes are compatible if they have compatible names and labels
- names are compatible if they are the same or missing
- non-wildcard labels are compatible if they are the same
- 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**(*other*) → bool

Check if this axis 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 other axis is equal to this axis, 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

---

<code>Axis.to_hdf(filepath[, key])</code>	Write axis to a HDF file.
---	---------------------------

---

## larray.Axis.to\_hdf

`Axis.to_hdf(filepath, key=None) → None`

Write 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

<i>IGroup</i> (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\_\_**(key, name=None, axis=None)

#### Methods

<b>__init__</b> (key[, name, axis])	
<i>by</i> (length[, step, template])	Split group into several groups of specified length.
<i>containing</i> (substring)	Return a group with all the labels containing the specified substring.
<i>difference</i> (other)	Return (set) difference of this label group and other.
<i>endingwith</i> (suffix)	Return a group with the labels ending with the specified string.
<i>equals</i> (other)	Check if this group is equal to another group.
<i>eval</i> ()	Translate key to labels, if it is not already, expanding slices in the process.
<i>intersection</i> (other)	Return (set) intersection of this label group and other.
<i>matching</i> ([deprecated, pattern, regex])	Return a group with all the labels matching the specified pattern or regular expression.
<i>named</i> (name)	Return group with a different name.
<i>retarget_to</i> (target_axis)	Retarget group to another axis.
<i>set</i> ()	Create LSet from this group.
<i>startingwith</i> (prefix)	Return a group with the labels starting with the specified string.
<i>to_hdf</i> (filepath[, key, axis_key])	Write group to a HDF file.
<i>to_label</i> ()	Translate key to labels, if it is not already.
<i>translate</i> ([bound, stop])	compute position(s) of group.
<i>union</i> (other)	Return (set) union of this label group and other.
<i>with_axis</i> (axis)	Return group with a different axis.

## Attributes

<code>axis</code>
<code>format_string</code>
<code>key</code>
<code>name</code>

<code>IGroup.named(name)</code>	Return group with a different name.
<code>IGroup.with_axis(axis)</code>	Return group with a different axis.
<code>IGroup.by(length[, step, template])</code>	Split group into several groups of specified length.
<code>IGroup.equals(other)</code>	Check if this group is equal to another group.
<code>IGroup.translate([bound, stop])</code>	compute position(s) of group.
<code>IGroup.union(other)</code>	Return (set) union of this label group and other.
<code>IGroup.intersection(other)</code>	Return (set) intersection of this label group and other.
<code>IGroup.difference(other)</code>	Return (set) difference of this label group and other.
<code>IGroup.containing(substring)</code>	Return a group with all the labels containing the specified substring.
<code>IGroup.startingwith(prefix)</code>	Return a group with the labels starting with the specified string.
<code>IGroup.endingwith(suffix)</code>	Return a group with the labels ending with the specified string.
<code>IGroup.matching([deprecated, pattern, regex])</code>	Return a group with all the labels matching the specified pattern or regular expression.
<code>IGroup.to_hdf(filepath[, key, axis_key])</code>	Write group to a HDF file.

## `larray.IGroup.named`

`IGroup.named(name)` → Group

Return group with a different name.

### Parameters

#### **name**

[str] new name for group

### Returns

#### **Group**

## `larray.IGroup.with_axis`

`IGroup.with_axis(axis)` → `Group`

Return group with a different axis.

### Parameters

#### **axis**

[int, str, Axis] new axis for group

### Returns

**Group**

## `larray.IGroup.by`

`IGroup.by(length, step=None, template=None)` → `Tuple[Group]`

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(other) → bool`

Check 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(bound=None, stop=False) → Union[int, slice, Sequence[int]]`

compute position(s) of group.

**larray.IGroup.union**

`IGroup.union(other) → LSet`

Return (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(other) → LSet`

Return (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**(*other*) → *LSet*

Return (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.IGroup.containing

IGroup.**containing**(*substring*) → *LGroup*

Return 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.IGroup.startingwith

IGroup.**startingwith**(*prefix*) → *LGroup*

Return 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**(*suffix*) → *LGroup*

Return 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.LGroup.matching

IGroup.**matching**(*deprecated=None, pattern=None, regex=None*) → *LGroup*

Return 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**(filepath, key=None, axis\_key=None) → None

Write 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

<code>LGroup(key[, name, axis])</code>	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\_\_**(key, name=None, axis=None)

## Methods

<code>__init__(key[, name, axis])</code>	
<code>by(length[, step, template])</code>	Split group into several groups of specified length.
<code>containing(substring)</code>	Return a group with all the labels containing the specified substring.
<code>difference(other)</code>	Return (set) difference of this label group and other.
<code>endingwith(suffix)</code>	Return a group with the labels ending with the specified string.
<code>equals(other)</code>	Check if this group is equal to another group.
<code>eval()</code>	Translate key to labels, if it is not already, expanding slices in the process.
<code>intersection(other)</code>	Return (set) intersection of this label group and other.
<code>matching([deprecated, pattern, regex])</code>	Return a group with all the labels matching the specified pattern or regular expression.
<code>named(name)</code>	Return group with a different name.
<code>retarget_to(target_axis)</code>	Retarget group to another axis.
<code>set()</code>	Create LSet from this group.
<code>startingwith(prefix)</code>	Return a group with the labels starting with the specified string.
<code>to_hdf(filepath[, key, axis_key])</code>	Write group to a HDF file.
<code>to_label()</code>	Translate key to labels, if it is not already.
<code>translate([bound, stop])</code>	compute position(s) of group.
<code>union(other)</code>	Return (set) union of this label group and other.
<code>with_axis(axis)</code>	Return group with a different axis.

## Attributes

<code>axis</code>
<code>format_string</code>
<code>key</code>
<code>name</code>

<code>LGroup.named(name)</code>	Return group with a different name.
<code>LGroup.with_axis(axis)</code>	Return group with a different axis.
<code>LGroup.by(length[, step, template])</code>	Split group into several groups of specified length.
<code>LGroup.equals(other)</code>	Check if this group is equal to another group.
<code>LGroup.translate([bound, stop])</code>	compute position(s) of group.
<code>LGroup.union(other)</code>	Return (set) union of this label group and other.
<code>LGroup.intersection(other)</code>	Return (set) intersection of this label group and other.
<code>LGroup.difference(other)</code>	Return (set) difference of this label group and other.
<code>LGroup.containing(substring)</code>	Return a group with all the labels containing the specified substring.
<code>LGroup.startingwith(prefix)</code>	Return a group with the labels starting with the specified string.
<code>LGroup.endingwith(suffix)</code>	Return a group with the labels ending with the specified string.
<code>LGroup.matching([deprecated, pattern, regex])</code>	Return a group with all the labels matching the specified pattern or regular expression.
<code>LGroup.to_hdf(filepath[, key, axis_key])</code>	Write group to a HDF file.

## **larray.LGroup.named**

`LGroup.named(name)` → Group

Return group with a different name.

### **Parameters**

**name**

[str] new name for group

### **Returns**

**Group**

## **larray.LGroup.with\_axis**

`LGroup.with_axis(axis)` → Group

Return group with a different axis.

### **Parameters**

**axis**

[int, str, Axis] new axis for group

### **Returns**

**Group**



## `larray.LGroup.by`

`LGroup.by(length, step=None, template=None) → Tuple[Group]`

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(other) → bool`

Check 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(bound=None, stop=False)` → `int`  
compute position(s) of group.

## `larray.LGroup.union`

`LGroup.union(other)` → `LSet`

Return (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**(*other*) → *LSet*

Return (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(other) → LSet`

Return (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(substring) → LGroup`

Return 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**(*prefix*) → *LGroup*

Return 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.LGroup.endingwith****LGroup.endingwith**(*suffix*) → *LGroup*

Return 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.LGroup.matching`

`LGroup.matching(deprecated=None, pattern=None, regex=None) → LGroup`

Return 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(filepath, key=None, axis_key=None) → None`

Write 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\_\_**(*key*, *name=None*, *axis=None*)



## Methods

<code>__init__(key[, name, axis])</code>	
<code>by(length[, step, template])</code>	Split group into several groups of specified length.
<code>containing(substring)</code>	Return a group with all the labels containing the specified substring.
<code>difference(other)</code>	Return (set) difference of this label group and other.
<code>endingwith(suffix)</code>	Return a group with the labels ending with the specified string.
<code>equals(other)</code>	Check if this group is equal to another group.
<code>eval()</code>	Translate key to labels, if it is not already, expanding slices in the process.
<code>intersection(other)</code>	Return (set) intersection of this label group and other.
<code>matching([deprecated, pattern, regex])</code>	Return a group with all the labels matching the specified pattern or regular expression.
<code>named(name)</code>	Return group with a different name.
<code>retarget_to(target_axis)</code>	Retarget group to another axis.
<code>set()</code>	Create LSet from this group.
<code>startingwith(prefix)</code>	Return a group with the labels starting with the specified string.
<code>to_hdf(filepath[, key, axis_key])</code>	Write group to a HDF file.
<code>to_label()</code>	Translate key to labels, if it is not already.
<code>translate([bound, stop])</code>	compute position(s) of group.
<code>union(other)</code>	Return (set) union of this label group and other.
<code>with_axis(axis)</code>	Return 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\_\_**(*axes=None*)**Methods**

---

<code><i>__init__</i>([axes])</code>	
<code><i>align</i>(other[, join, axes])</code>	Align this axis collection with another.
<code><i>append</i>(axis)</code>	Append axis at the end of the collection.
<code><i>axis_id</i>(axis)</code>	Return the id of an axis.
<code><i>check_compatible</i>(axes)</code>	Check if axes passed as argument are compatible with those contained in the collection.
<code><i>combine_axes</i>([axes, sep, wildcard, ...])</code>	Combine several axes into one.
<code><i>copy</i>()</code>	Return a copy.
<code><i>extend</i>(axes[, validate, replace_wildcards])</code>	Extend the collection by appending the axes from <i>axes</i> .
<code><i>get</i>(key[, default, name])</code>	Return axis corresponding to key.
<code><i>get_all</i>(key)</code>	Return all axes from key if present and length 1 wildcard axes otherwise.
<code><i>get_by_pos</i>(key, i)</code>	Return axis corresponding to a key, or to position i if the key has no name and key object not found.
<code><i>index</i>(axis[, compatible])</code>	Return the index of axis.
<code><i>insert</i>(index, axis)</code>	Insert axis before index.
<code><i>isaxis</i>(value)</code>	Test if input is an Axis object or the name of an axis contained in self.
<code><i>iter_labels</i>([axes, ascending])</code>	Return a view of the axes labels.
<code><i>keys</i>()</code>	Return list of all axis names.
<code><i>pop</i>([axis])</code>	Remove and return an axis.
<code><i>rename</i>([renames, to])</code>	Rename axes of the collection.
<code><i>replace</i>([axes_to_replace, new_axis, inplace])</code>	Replace one, several or all axes of the collection.
<code><i>set_labels</i>([axis, labels, inplace])</code>	Replace the labels of one or several axes.
<code><i>split_axes</i>([axes, sep, names, regex])</code>	Split axes and returns a new collection.
<code><i>split_axis</i>(**kwargs)</code>	
<code><i>union</i>(*args[, validate, replace_wildcards])</code>	
<code><i>without</i>(axes)</code>	Return a new collection without some axes.

---

## Attributes

<i>display_names</i>	Return the list of (display) names of the axes.
<i>ids</i>	Return the list of ids of the axes.
<i>info</i>	Describe the collection (shape and labels for each axis).
<i>labels</i>	Return the list of labels of the axes.
<i>names</i>	Return the list of (raw) names of the axes.
<i>ndim</i>	
<i>shape</i>	Return the shape of the collection.
<i>size</i>	Return the size of the collection, i.e. the number of elements of the array.

<i>AxisCollection.names</i>	Return the list of (raw) names of the axes.
<i>AxisCollection.display_names</i>	Return the list of (display) names of the axes.
<i>AxisCollection.labels</i>	Return the list of labels of the axes.
<i>AxisCollection.shape</i>	Return the shape of the collection.
<i>AxisCollection.size</i>	Return the size of the collection, i.e. the number of elements of the array.
<i>AxisCollection.info</i>	Describe the collection (shape and labels for each axis).
<i>AxisCollection.copy()</i>	Return a copy.

## larray.AxisCollection.names

**property** AxisCollection.names: List[str]

Return 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: List[str]

Return 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: List[Iterable[Union[bool, int, float, str, bytes, generic]]]

Return 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: Tuple[int, ...]

Return 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: **int**

Return 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: **str**

Describe 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**() → *AxisCollection*

Return a copy.

## Searching

<i>AxisCollection.keys()</i>	Return list of all axis names.
<i>AxisCollection.index</i> (axis[, compatible])	Return the index of axis.
<i>AxisCollection.axis_id</i> (axis)	Return the id of an axis.
<i>AxisCollection.ids</i>	Return the list of ids of the axes.
<i>AxisCollection.iter_labels</i> ([axes, ascending])	Return a view of the axes labels.

## larray.AxisCollection.keys

AxisCollection.**keys**() → List[str]

Return 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**(axis, compatible=False) → int

Return 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**(axis) → Union[str, int]

Return 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**: List[Union[str, int]]

Return 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(axes=None, ascending=True) → Sequence[Tuple[IGroup, ...]]`

Return 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 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])
```

(continues on next page)



(continued from previous page)

```

>>> 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(key[, default, name])</code>	Return axis corresponding to key.
<code>AxisCollection.get_by_pos(key, i)</code>	Return 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(key)</code>	Return all axes from key if present and length 1 wildcard axes otherwise.
<code>AxisCollection.pop([axis])</code>	Remove and return an axis.
<code>AxisCollection.append(axis)</code>	Append axis at the end of the collection.
<code>AxisCollection.extend(axes[, validate, ...])</code>	Extend the collection by appending the axes from <i>axes</i> .
<code>AxisCollection.insert(index, axis)</code>	Insert axis before index.
<code>AxisCollection.rename([renames, to])</code>	Rename axes of the collection.
<code>AxisCollection.replace([axes_to_replace, ...])</code>	Replace one, several or all axes of the collection.
<code>AxisCollection.set_labels([axis, labels, ...])</code>	Replace the labels of one or several axes.
<code>AxisCollection.without(axes)</code>	Return a new collection without some axes.
<code>AxisCollection.combine_axes([axes, sep, ...])</code>	Combine several axes into one.
<code>AxisCollection.split_axes([axes, sep, ...])</code>	Split axes and returns a new collection.
<code>AxisCollection.align(other[, join, axes])</code>	Align this axis collection with another.

## larray.AxisCollection.get

`AxisCollection.get(key, default=None, name=None) → Axis`

Return 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(key, i) → Axis`

Return 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(key) → AxisCollection`

Return 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**(axis=-1) → *Axis*

Remove and return 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(axis) → None`

Append 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(axes, validate=True, replace_wildcards=False) → None`

Extend 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**(*index*, *axis*) → None

Insert 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'),
  Axis([2007, 2008, 2009, 2010], 'time')
])
```

## `larray.AxisCollection.rename`

`AxisCollection.rename(renames=None, to=None, **kwargs) → AxisCollection`

Rename axes of the collection.

### Parameters

#### **renames**

[axis ref or dict {axis ref: str} or list of tuple (axis ref, str), optional] Rename 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,F0')
>>> sex = Axis('sex=M,F')
>>> axes = AxisCollection([nat, sex])
>>> axes
AxisCollection([
  Axis(['BE', 'F0'], 'nat'),
  Axis(['M', 'F'], 'sex')
])
>>> axes.rename(nat, 'nat2')
AxisCollection([
  Axis(['BE', 'F0'], 'nat2'),
  Axis(['M', 'F'], 'sex')
])
>>> axes.rename(nat='nat2', sex='sex2')
AxisCollection([
  Axis(['BE', 'F0'], 'nat2'),
  Axis(['M', 'F'], 'sex2')
])
>>> axes.rename([(nat, 'nat2'), (sex, 'sex2')])
AxisCollection([
  Axis(['BE', 'F0'], 'nat2'),
  Axis(['M', 'F'], 'sex2')
])
>>> axes.rename({'nat': 'nat2', 'sex': 'sex2'})
AxisCollection([
  Axis(['BE', 'F0'], 'nat2'),
  Axis(['M', 'F'], 'sex2')
])
```

**larray.AxisCollection.replace**

**AxisCollection.replace**(*axes\_to\_replace=None*, *new\_axis=None*, *inplace=False*, *\*\*kwargs*) → *AxisCollection*

Replace one, several or all axes of the collection.

**Parameters****axes\_to\_replace**

[axis ref or dict {axis ref: axis} or list of (tuple or 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. If a list of tuple is given, it must be pairs of (reference to old axis, new axis).

**new\_axis**

[axis ref, optional] New axis if *axes\_to\_replace* contains a single axis reference. Defaults to None.

**inplace**

[bool, optional] Whether 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([
  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**(axis=None, labels=None, inplace=False, \*\*kwargs) → *AxisCollection*

Replace 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 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.

**Warning:** Not passing a mapping but the complete list of new labels as the ‘labels’ argument must be done with caution. Make sure that the order of new labels corresponds to the exact same order of previous 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([
  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'),
```

(continues on next page)

(continued from previous page)

```
    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')
])
>>> axes.set_labels({'M': 'Men', 'F': 'Women'})
AxisCollection([
    Axis(['BE', 'FO'], 'nat'),
    Axis(['Men', 'Women'], 'sex')
])
```

## **larray.AxisCollection.without**

**AxisCollection.without**(*axes*) → *AxisCollection*

Return 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(axes=None, sep='_', wildcard=False, front_if_spread=False) → AxisCollection`

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 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 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(axes=None, sep='_', names=None, regex=None) → AxisCollection`

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()
AxisCollection([
  Axis(['a0', 'a1'], 'a'),
  Axis(['b0', 'b1', 'b2'], 'b')
])
```

Split labels using a regular expression

```
>>> combined = AxisCollection('ab = a0b0..a1b2')
>>> combined
AxisCollection([
  Axis(['a0b0', 'a0b1', 'a0b2', 'a1b0', 'a1b1', 'a1b2'], 'ab')
])
>>> # The pattern for each resulting axis should be enclosed in parentheses
>>> combined.split_axes('ab', names=['a', 'b'], regex=r'(..)(..)')
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**(other, join='outer', axes=None) → **Tuple**[**AxisCollection**, **AxisCollection**]

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)
```

(continues on next page)

(continued from previous page)

```

])
>>> aligned1, aligned2 = col1.align(col2)
>>> aligned1
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(value)</code>	Test if input is an Axis object or the name of an axis contained in self.
<code>AxisCollection.check_compatible(axes)</code>	Check if axes passed as argument are compatible with those contained in the collection.

## larray.AxisCollection.isaxis

`AxisCollection.isaxis(value) → bool`

Test 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(axes) → None`

Check 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**


---

`Array(data[, axes, title, meta, dtype])`

---

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 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
```

(continues on next page)

(continued from previous page)

```
>>> # with metadata
>>> 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
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] Return metadata of the array.

**\_\_init\_\_**(data, axes=None, title=None, meta=None, dtype=None)

## Methods

<code>__init__(data[, axes, title, meta, dtype])</code>	
<code>align(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, keepaxes])</code>	Test whether all selected elements evaluate to True.
<code>allclose(other[, rtol, atol, nans_equal, ...])</code>	Compare this array with another array and returns True if they are element-wise equal within a tolerance.
<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, keepaxes])</code>	Test whether any selected elements evaluate to True.
<code>append(axis, value[, label])</code>	Add a value to this array along an axis.
<code>apply(transform, *args[, by, axes, dtype, ...])</code>	Apply a transformation function to array elements.
<code>apply_map(mapping[, dtype])</code>	Apply a transformation mapping to array elements.
<code>argmax(**kwargs)</code>	
<code>argmin(**kwargs)</code>	
<code>argsort(**kwargs)</code>	
<code>as_table(**kwargs)</code>	
<code>astype(dtype[, order, casting, subok, copy])</code>	Copy of the array, cast to a specified type.
<code>broadcast_with(target[, check_compatible])</code>	Return an array that is (NumPy) broadcastable with target.
<code>clip([minval, maxval, out])</code>	Clip (limit) the values in an array.
<code>combine_axes([axes, sep, wildcard])</code>	Combine several axes into one.
<code>compact([display, name])</code>	Detect and remove "useless" axes (ie axes for which values are constant over the whole axis).
<code>copy()</code>	Return a copy of the array.
<code>cumprod([axis])</code>	Return the cumulative product of array elements.
<code>cumsum([axis])</code>	Return the cumulative sum of array elements along an axis.
<code>describe(*args[, percentiles])</code>	Descriptive summary statistics, excluding NaN values.
<code>describe_by(*args[, percentiles])</code>	Descriptive summary statistics, excluding NaN values, along axes or for groups.
<code>diff([axis, d, n, label])</code>	Compute the n-th order discrete difference along a given axis.
<code>divnot0(other)</code>	Divide this array by other, but return 0.0 where other is 0.
<code>drop([labels])</code>	Return array without some labels or indices along an axis.
<code>drop_labels(**kwargs)</code>	
<code>dump(self[, header, wide, value_name, ...])</code>	Dump array as a 2D nested list.
<code>eq(other[, rtol, atol, nans_equal])</code>	Compare this array with another array element-wise and returns an array of booleans.

continues on next page

Table 2 – continued from previous page

<i>equals</i> (other[, rtol, atol, nans_equal, ...])	Compare this array with another array and returns True if they have the same axes and elements, False otherwise.
<i>expand</i> ([target_axes, out, readonly])	Expand this array to target_axes.
<i>extend</i> (**kwargs)	
<i>filter</i> ([collapse])	Filter the array along the axes given as keyword arguments.
<i>growth_rate</i> ([axis, d, label])	Compute the growth along a given axis.
<i>ignore_labels</i> ([axes])	Ignore labels from axes (replace those axes by "wild-card" axes).
<i>indexofmax</i> ([axis])	Return indices of the maximum values along a given axis.
<i>indexofmin</i> ([axis])	Return indices of the minimum values along a given axis.
<i>indicesofsorted</i> ([axis, ascending, kind])	Return the indices that would sort this array.
<i>insert</i> (value[, before, after, pos, axis, label])	Insert value in array along an axis.
<i>isin</i> (test_values[, assume_unique, invert])	Compute whether each element of this array is in <i>test_values</i> .
<i>items</i> ([axes, ascending])	Return a (label, value) view of the array along axes.
<i>keys</i> ([axes, ascending])	Return a view on the array labels along axes.
<i>labelofmax</i> ([axis])	Return labels of the maximum values along a given axis.
<i>labelofmin</i> ([axis])	Return labels of the minimum values along a given axis.
<i>labelsofsorted</i> ([axis, ascending, kind])	Return 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.
<i>max_by</i> (*axes_and_groups[, out, skipna, keepaxes])	Get maximum of array elements for the given axes/groups.
<i>mean</i> (*axes_and_groups[, dtype, out, skipna, ...])	Compute the arithmetic mean.
<i>mean_by</i> (*axes_and_groups[, dtype, out, ...])	Compute the arithmetic mean.
<i>median</i> (*axes_and_groups[, out, skipna, keepaxes])	Compute the arithmetic median.
<i>median_by</i> (*axes_and_groups[, out, skipna, ...])	Compute the arithmetic median.
<i>min</i> (*axes_and_groups[, out, skipna, keepaxes])	Get minimum of array elements along given axes/groups.
<i>min_by</i> (*axes_and_groups[, out, skipna, keepaxes])	Get minimum of array elements for the given axes/groups.
<i>nonzero</i> ()	Return the indices of the elements that are non-zero.
<i>percent</i> (*axes)	Return an array with values given as percent of the total of all values along given axes.
<i>percentile</i> (q, *axes_and_groups[, out, ...])	Compute the qth percentile of the data along the specified axis.
<i>percentile_by</i> (q, *axes_and_groups[, out, ...])	Compute the qth percentile of the data for the specified axis.
<i>posargmax</i> (**kwargs)	
<i>posargmin</i> (**kwargs)	
<i>posargsort</i> (**kwargs)	

continues on next page

Table 2 – continued from previous page

<code>prepend(axis, value[, label])</code>	Add an array before this array along an axis.
<code>prod(*axes_and_groups[, dtype, out, skipna, ...])</code>	Compute the product of array elements along given axes/groups.
<code>prod_by(*axes_and_groups[, dtype, out, ...])</code>	Compute the product of array elements for the given axes/groups.
<code>ptp(*axes_and_groups[, out])</code>	Return the range of values (maximum - minimum).
<code>ratio(*axes)</code>	Return an array with all values divided by the sum of values along given axes.
<code>rationot0(*axes)</code>	Return an Array with values array / array.sum(axes) where the sum is not 0, 0 otherwise.
<code>reindex([axes_to_reindex, new_axis, ...])</code>	Reorder and/or add new labels in axes.
<code>rename([renames, to, inplace])</code>	Rename axes of the array.
<code>reshape(target_axes)</code>	Given a list of new axes, changes the shape of the array.
<code>reshape_like(target)</code>	Same as reshape but with an array as input.
<code>reverse([axes])</code>	Reverse axes of an array.
<code>roll([axis, n])</code>	Roll the cells of the array n-times to the right along axis.
<code>set(value, **kwargs)</code>	Set a subset of array to value.
<code>set_axes([axes_to_replace, new_axis, inplace])</code>	Replace one, several or all axes of the array.
<code>set_labels([axis, labels, inplace])</code>	Replace the labels of one or several axes of the array.
<code>shift(axis[, n])</code>	Shift the cells of the array n-times to the right along axis.
<code>sort_axes(**kwargs)</code>	
<code>sort_axis(**kwargs)</code>	
<code>sort_labels([axes, ascending])</code>	Sort labels of axes of the array.
<code>sort_values([key, axis, ascending])</code>	Sort values of the array.
<code>split_axes([axes, sep, names, regex, sort, ...])</code>	Split axes and returns a new array.
<code>split_axis(**kwargs)</code>	
<code>std(*axes_and_groups[, dtype, ddof, out, ...])</code>	Compute the sample standard deviation.
<code>std_by(*axes_and_groups[, dtype, ddof, out, ...])</code>	Compute the sample standard deviation.
<code>sum(*axes_and_groups[, dtype, out, skipna, ...])</code>	Compute the sum of array elements along given axes/groups.
<code>sum_by(*axes_and_groups[, dtype, out, ...])</code>	Compute the sum of array elements for the given axes/groups.
<code>to_clipboard(*args, **kwargs)</code>	Send the content of the array to the clipboard.
<code>to_csv(filepath[, sep, na_rep, wide, ...])</code>	Write array to a csv file.
<code>to_excel([filepath, sheet, position, ...])</code>	Write array in the specified sheet of specified excel workbook.
<code>to_frame([fold_last_axis_name, dropna])</code>	Convert an Array into a Pandas DataFrame.
<code>to_hdf(filepath, key)</code>	Write array to a HDF file.
<code>to_series([name, dropna])</code>	Convert an Array into a Pandas Series.
<code>to_stata(filepath_or_buffer, **kwargs)</code>	Write array to a Stata .dta file.
<code>transpose(*args)</code>	Reorder axes.
<code>unique([axes, sort, sep])</code>	Return unique values (optionally along axes).
<code>value_counts()</code>	Count number of occurrences of each unique value in array.
<code>values([axes, ascending])</code>	Return a view on the values of the array along axes.

continues on next page

Table 2 – continued from previous page

<i>var</i> (*axes_and_groups[, dtype, ddof, out, ...])	Compute the unbiased variance.
<i>var_by</i> (*axes_and_groups[, dtype, ddof, out, ...])	Compute the unbiased variance.
<i>with_axes</i> (**kwargs)	
<i>with_total</i> (*args[, op, label])	Add aggregated values (sum by default) along each axis.

## Attributes

<i>data</i>	
<i>axes</i>	
<i>T</i>	Reorder axes.
<i>df</i>	Convert an Array into a Pandas DataFrame.
<i>dtype</i>	Return the type of the data of the array.
<i>i</i>	Allows selection of a subset using indices of labels.
<i>iflat</i>	Access the array by index as if it was flat (one dimensional) and all its axes were combined.
<i>info</i>	Describe an Array (metadata + shape and labels for each axis).
<i>ipoints</i>	Allows selection of arbitrary items in the array based on their N-dimensional index.
<i>item</i>	
<i>memory_used</i>	Return the memory consumed by the array in human readable form.
<i>meta</i>	Return metadata of the array.
<i>nbytes</i>	Return the number of bytes used to store the array in memory.
<i>ndim</i>	Return the number of dimensions of the array.
<i>plot</i>	Plot the data of the array into a graph (window pop-up).
<i>points</i>	Allows selection of arbitrary items in the array based on their N-dimensional label index.
<i>series</i>	Convert an Array into a Pandas Series.
<i>shape</i>	Return the shape of the array as a tuple.
<i>size</i>	Return the number of elements in array.
<i>title</i>	

## Array Creation Functions

<code>sequence</code> (axis[, initial, inc, mult, func, ...])	Create an array by sequentially applying modifications to the array along axis.
<code>ndtest</code> (shape_or_axes[, start, label_start, ...])	Return test array with given shape.
<code>zeros</code> (axes[, title, dtype, order, meta])	Return an array with the specified axes and filled with zeros.
<code>zeros_like</code> (array[, title, dtype, order, meta])	Return an array with the same axes as array and filled with zeros.
<code>ones</code> (axes[, title, dtype, order, meta])	Return an array with the specified axes and filled with ones.
<code>ones_like</code> (array[, title, dtype, order, meta])	Return an array with the same axes as array and filled with ones.
<code>empty</code> (axes[, title, dtype, order, meta])	Return an array with the specified axes and uninitialized (arbitrary) data.
<code>empty_like</code> (array[, title, dtype, order, meta])	Return an array with the same axes as array and uninitialized (arbitrary) data.
<code>full</code> (axes, fill_value[, title, dtype, ...])	Return an array with the specified axes and filled with fill_value.
<code>full_like</code> (array, fill_value[, title, dtype, ...])	Return 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=None, func=None, axes=None, title=None, meta=None) → *Array*

Create 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 1 unless mult is provided (in which case it defaults to 0).

##### **mult**

[scalar, Array, optional] Value to multiply the previous value by. Defaults to None.

##### **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 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)
```

(continues on next page)

(continued from previous page)

```

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
      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) → Array`

Return 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 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,F0;sex=M,F")
nat\sex M F
      BE 0 1
      FO 2 3
>>> nat = Axis("nat=BE,F0")
>>> 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) → Array`

Return 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 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) → Array`

Return 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 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) → Array`

Return 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 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) → Array`

Return 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 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) → Array`

Return 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 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,F0')
>>> 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) → Array`

Return 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 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) → Array`

Return 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 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
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) → Array`

Return 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 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()</code>	Return 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()` → *Array*

Return 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(t)`.

**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)

<code>Array.info</code>	Describe an Array (metadata + shape and labels for each axis).
<code>Array.shape</code>	Return the shape of the array as a tuple.
<code>Array.ndim</code>	Return the number of dimensions of the array.
<code>Array.dtype</code>	Return the type of the data of the array.
<code>Array.size</code>	Return the number of elements in array.
<code>Array.nbytes</code>	Return the number of bytes used to store the array in memory.
<code>Array.memory_used</code>	Return the memory consumed by the array in human readable form.

## `larray.Array.info`

**property** `Array.info`: `str`

Describe 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,F0; 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`: `Tuple[int, ...]`

Return the shape of the array as a tuple.

### Returns

`tuple`

Tuple representing the current shape.

## Examples

```
>>> a = ndtest('nat=BE,F0;sex=M,F;type=type1,type2,type3')
>>> a.shape
(2, 2, 3)
```

## larray.Array.ndim

**property** Array.ndim: **int**

Return the number of dimensions of the array.

### Returns

**int**

Number of dimensions of an Array.

## Examples

```
>>> a = ndtest('nat=BE,F0;sex=M,F')
>>> a.ndim
2
```

## larray.Array.dtype

**property** Array.dtype: **dtype**

Return 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: **int**

Return 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`: `int`

Return 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`: `str`

Return 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(value, **kwargs)</code>	Set a subset of array to value.
<code>Array.drop([labels])</code>	Return array without some labels or indices along an axis.
<code>Array.ignore_labels([axes])</code>	Ignore labels from axes (replace those axes by "wild-card" axes).
<code>Array.filter([collapse])</code>	Filter the array along the axes given as keyword arguments.
<code>Array.apply(transform, *args[, by, axes, ...])</code>	Apply a transformation function to array elements.
<code>Array.apply_map(mapping[, dtype])</code>	Apply a transformation mapping to array elements.

## larray.Array.i

### Array.i

Allows selection of a subset using indices of labels.

### Notes

Using `.i[]` is equivalent to numpy indexing when indexing along a single axis. However, when indexing along multiple axes this indexes the cross product instead of points.

### 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):
...
ValueError: all combined keys should have the same length
```

## 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):
...
ValueError: all combined keys should have the same length
```

```
>>> arr.ipoints[[0, 1], [0, 2]]
a_b\c  c0  c1  c2  c3
a0_b0   0   1   2   3
a1_b2  20  21  22  23
```

## 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(value, **kwargs) → None`

Set 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(labels=None) → Array`

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**(*axes=None*) → *Array*

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
```

(continues on next page)

(continued from previous page)

```

>>> arr1.ignore_labels([a, b])
a*\b*  0  1
      0  0  1
      1  2  3
>>> arr2 = ndtest([a, b2])
>>> 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(collapse=False, **kwargs) → Array`

Filter 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(transform, *args, by=None, axes=None, dtype=None, ascending=True, **kwargs) → Union[Array, bool, int, float, str, bytes, generic, Tuple[Array, ...]]`

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 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')
```

(continues on next page)

(continued from previous page)

```
... 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(mapping, dtype=None) → Union[Array, bool, int, float, str, bytes, generic, Tuple[Array, ...]]`

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([axes_to_replace, new_axis, ...])</code>	Replace one, several or all axes of the array.
<code>Array.rename([renames, to, inplace])</code>	Rename axes of the array.
<code>Array.set_labels([axis, labels, inplace])</code>	Replace the labels of one or several axes of the array.
<code>Array.combine_axes([axes, sep, wildcard])</code>	Combine several axes into one.
<code>Array.split_axes([axes, sep, names, regex, ...])</code>	Split axes and returns a new array.
<code>Array.reverse([axes])</code>	Reverse axes of an array.

### larray.Array.set\_axes

`Array.set_axes(axes_to_replace=None, new_axis=None, inplace=False, **kwargs) → Array`

Replace one, several or all axes of the array.

#### Parameters

##### **axes\_to\_replace**

[axis ref or dict {axis ref: axis} or list of (tuple or 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. If a list of tuple is given, it must be pairs of (reference to old axis, new axis).

##### **new\_axis**

[Axis, optional] New axis if *axes\_to\_replace* contains a single axis reference.

##### **inplace**

[bool, optional] Whether 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**(*renames=None, to=None, inplace=False, \*\*kwargs*) → *Array*

Rename axes of the array.

### **Parameters**

#### **renames**

[axis ref or dict {axis ref: str} or list of tuple (axis ref, str)] Rename 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(axis=None, labels=None, inplace=False, **kwargs) → Array`

Replace 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 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.

**Warning:** Not passing a mapping but the complete list of new labels as the 'labels' argument must be done with caution. Make sure that the order of new labels corresponds to the exact same order of previous labels.

See also:

[\*AxisCollection.set\\_labels\*](#)

## Examples

```
>>> a = ndtest('nat=BE,F0;sex=M,F')
>>> a
nat\sex  M  F
      BE  0  1
      F0  2  3
>>> a.set_labels('sex', ['Men', 'Women'])
nat\sex  Men  Women
      BE    0    1
      F0    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
      F0    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
      F0    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(axes=None, sep='_', wildcard=False) → Array`

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 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

```

(continues on next page)

(continued from previous page)

```
>>> 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
```

## larray.Array.split\_axes

`Array.split_axes(axes=None, sep='_', names=None, regex=None, sort=False, fill_value=nan) → Array`

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 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()
```

(continues on next page)

(continued from previous page)

```
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})')
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.reverse****Array.reverse**(*axes=None*) → *Array*

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>	Compute the sum of array elements along given axes/groups.
<code>Array.sum_by(*axes_and_groups[, dtype, out, ...])</code>	Compute the sum of array elements for the given axes/groups.
<code>Array.prod(*axes_and_groups[, dtype, out, ...])</code>	Compute the product of array elements along given axes/groups.
<code>Array.prod_by(*axes_and_groups[, dtype, ...])</code>	Compute the product of array elements for the given axes/groups.
<code>Array.cumsum([axis])</code>	Return the cumulative sum of array elements along an axis.
<code>Array.cumprod([axis])</code>	Return the cumulative product of array elements.
<code>Array.mean(*axes_and_groups[, dtype, out, ...])</code>	Compute the arithmetic mean.
<code>Array.mean_by(*axes_and_groups[, dtype, ...])</code>	Compute the arithmetic mean.
<code>Array.median(*axes_and_groups[, out, ...])</code>	Compute the arithmetic median.
<code>Array.median_by(*axes_and_groups[, out, ...])</code>	Compute the arithmetic median.
<code>Array.var(*axes_and_groups[, dtype, ddof, ...])</code>	Compute the unbiased variance.
<code>Array.var_by(*axes_and_groups[, dtype, ...])</code>	Compute the unbiased variance.
<code>Array.std(*axes_and_groups[, dtype, ddof, ...])</code>	Compute the sample standard deviation.
<code>Array.std_by(*axes_and_groups[, dtype, ...])</code>	Compute the sample standard deviation.
<code>Array.percentile(q, *axes_and_groups[, out, ...])</code>	Compute the qth percentile of the data along the specified axis.
<code>Array.percentile_by(q, *axes_and_groups[, ...])</code>	Compute the qth percentile of the data for the specified axis.
<code>Array.ptp(*axes_and_groups[, out])</code>	Return 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(*axes)</code>	Return an array with values given as percent of the total of all values along given axes.
<code>Array.ratio(*axes)</code>	Return an array with all values divided by the sum of values along given axes.
<code>Array.rationot0(*axes)</code>	Return an Array with values array / array.sum(axes) where the sum is not 0, 0 otherwise.
<code>Array.growth_rate([axis, d, label])</code>	Compute the growth along a given axis.
<code>Array.describe(*args[, percentiles])</code>	Descriptive summary statistics, excluding NaN values.
<code>Array.describe_by(*args[, percentiles])</code>	Descriptive summary statistics, excluding NaN values, along axes or for groups.
<code>Array.value_counts()</code>	Count number of occurrences of each unique value in array.

## larray.Array.sum

`Array.sum(*axes_and_groups, dtype=None, out=None, skipna=None, keepaxes=False, **explicit_axes)`

Compute 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 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 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)`

Compute 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 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 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)`

Compute 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 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 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)`

Compute 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 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 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(axis=-1) → Union[Array, bool, int, float, str, bytes, generic]`

Return 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(axis=-1) → Union[Array, bool, int, float, str, bytes, generic]`

Return 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)`

Compute 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 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 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)`

Compute 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 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 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)`

Compute 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 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 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)`

Compute 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 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 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)

Compute 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 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 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:')
a\b   b0,b1,b3;b5:
a0         9.0   1.0
a1         4.0   1.0
```

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)`

Compute 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 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 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
```

(continues on next page)



(continued from previous page)

```

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)`

Compute 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 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 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)

Compute 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 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 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
```

(continues on next page)

(continued from previous page)

```

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*, *method='linear'*, *skipna=None*, *keepaxes=False*, *\*\*explicit\_axes*)

Compute the *q*th 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 *q*th percentile is performed. The default (no axis or group) is to perform the *q*th 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 *q*th 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).

**method**

[str, optional] This parameter specifies the method to use for estimating the percentile when the desired percentile lies between two indexes. The different methods supported are described in the Notes section. The options are:

- 'inverted\_cdf'
- 'averaged\_inverted\_cdf'
- 'closest\_observation'
- 'interpolated\_inverted\_cdf'
- 'hazen'
- 'weibull'
- 'linear' (default)
- 'median\_unbiased'
- 'normal\_unbiased'
- 'lower'
- 'higher'
- 'midpoint'
- 'nearest'

The first three and last four methods are discontinuous. Defaults to 'linear'.

**skipna**

[bool, optional] Whether 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 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\*](#)

## Notes

Given a vector  $V$  of length  $n$ , the  $q$ -th percentile of  $V$  is the value  $q/100$  of the way from the minimum to the maximum in a sorted copy of  $V$ . The values and distances of the two nearest neighbors as well as the *method* parameter will determine the percentile if the normalized ranking does not match the location of  $q$  exactly. This function is the same as the median if  $q=50$ , the same as the minimum if  $q=0$  and the same as the maximum if  $q=100$ .

The optional *method* parameter specifies the method to use when the desired percentile lies between two indexes  $i$  and  $j = i + 1$ . In that case, we first determine  $i + g$ , a virtual index that lies between  $i$  and  $j$ , where  $i$  is the floor and  $g$  is the fractional part of the index. The final result is, then, an interpolation of  $a[i]$  and  $a[j]$  based on  $g$ . During the computation of  $g$ ,  $i$  and  $j$  are modified using correction constants  $\alpha$  and  $\beta$  whose choices depend on the *method* used. Finally, note that since Python uses 0-based indexing, the code subtracts another 1 from the index internally.

The following formula determines the virtual index  $i + g$ , the location of the percentile in the sorted sample:

$$i + g = (q/100) * (n - \alpha - \beta + 1) + \alpha$$

The different methods then work as follows

### **inverted\_cdf:**

method 1 of H&F [1]. This method gives discontinuous results:

- if  $g > 0$  ; then take  $j$
- if  $g = 0$  ; then take  $i$

### **averaged\_inverted\_cdf:**

method 2 of H&F [1]. This method give discontinuous results:

- if  $g > 0$  ; then take  $j$
- if  $g = 0$  ; then average between bounds

### **closest\_observation:**

method 3 of H&F [1]. This method give discontinuous results:

- if  $g > 0$  ; then take  $j$
- if  $g = 0$  and index is odd ; then take  $j$
- if  $g = 0$  and index is even ; then take  $i$

### **interpolated\_inverted\_cdf:**

method 4 of H&F [1]. This method give continuous results using:

- $\alpha = 0$
- $\beta = 1$

### **hazen:**

method 5 of H&F [1]. This method give continuous results using:

- $\alpha = 1/2$
- $\beta = 1/2$

### **weibull:**

method 6 of H&F [1]. This method give continuous results using:

- $\alpha = 0$
- $\beta = 0$

**linear:**

method 7 of H&F [1]. This method give continuous results using:

- $\alpha = 1$
- $\beta = 1$

**median\_unbiased:**

method 8 of H&F [1]. This method is probably the best method if the sample distribution function is unknown (see reference). This method give continuous results using:

- $\alpha = 1/3$
- $\beta = 1/3$

**normal\_unbiased:**

method 9 of H&F [1]. This method is probably the best method if the sample distribution function is known to be normal. This method give continuous results using:

- $\alpha = 3/8$
- $\beta = 3/8$

**lower:**

NumPy method kept for backwards compatibility. Takes *i* as the interpolation point.

**higher:**

NumPy method kept for backwards compatibility. Takes *j* as the interpolation point.

**nearest:**

NumPy method kept for backwards compatibility. Takes *i* or *j*, whichever is nearest.

**midpoint:**

NumPy method kept for backwards compatibility. Uses  $(i + j) / 2$ .

## References

[1]

## 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
```

(continues on next page)



(continued from previous page)

```

    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
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*, *method='linear'*, *skipna=None*, *keepaxes=False*, *\*\*explicit\_axes*)

Compute 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).

**method**

[str, optional] This parameter specifies the method to use for estimating the percentile when the desired percentile lies between two indexes. The different methods supported are described in the Notes section. The options are:

- 'inverted\_cdf'
- 'averaged\_inverted\_cdf'
- 'closest\_observation'
- 'interpolated\_inverted\_cdf'
- 'hazen'
- 'weibull'
- 'linear' (default)
- 'median\_unbiased'
- 'normal\_unbiased'
- 'lower'
- 'higher'
- 'midpoint'
- 'nearest'

The first three and last four methods are discontinuous. Defaults to 'linear'.

**skipna**

[bool, optional] Whether 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 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\*](#)

**Notes**

Given a vector  $V$  of length  $n$ , the  $q$ -th percentile of  $V$  is the value  $q/100$  of the way from the minimum to the maximum in a sorted copy of  $V$ . The values and distances of the two nearest neighbors as well as the *method* parameter will determine the percentile if the normalized ranking does not match the location of  $q$  exactly. This function is the same as the median if  $q=50$ , the same as the minimum if  $q=0$  and the same as the maximum if  $q=100$ .

The optional *method* parameter specifies the method to use when the desired percentile lies between two indexes  $i$  and  $j = i + 1$ . In that case, we first determine  $i + g$ , a virtual index that lies between  $i$  and  $j$ , where  $i$  is the floor and  $g$  is the fractional part of the index. The final result is, then, an interpolation of  $a[i]$  and  $a[j]$  based on  $g$ . During the computation of  $g$ ,  $i$  and  $j$  are modified using correction constants  $\alpha$  and  $\beta$  whose choices depend on the method used. Finally, note that since Python uses 0-based indexing, the code subtracts another 1 from the index internally.

The following formula determines the virtual index  $i + g$ , the location of the percentile in the sorted sample:

$$i + g = (q/100) * (n - \alpha - \beta + 1) + \alpha$$

The different methods then work as follows

**inverted\_cdf:**

method 1 of H&F [1]. This method gives discontinuous results:

- if  $g > 0$  ; then take  $j$
- if  $g = 0$  ; then take  $i$

**averaged\_inverted\_cdf:**

method 2 of H&F [1]. This method give discontinuous results:

- if  $g > 0$  ; then take  $j$
- if  $g = 0$  ; then average between bounds

**closest\_observation:**

method 3 of H&F [1]. This method give discontinuous results:

- if  $g > 0$  ; then take  $j$
- if  $g = 0$  and index is odd ; then take  $j$

- if  $g = 0$  and index is even ; then take  $i$

**interpolated\_inverted\_cdf:**

method 4 of H&F [1]. This method give continuous results using:

- $\alpha = 0$
- $\beta = 1$

**hazen:**

method 5 of H&F [1]. This method give continuous results using:

- $\alpha = 1/2$
- $\beta = 1/2$

**weibull:**

method 6 of H&F [1]. This method give continuous results using:

- $\alpha = 0$
- $\beta = 0$

**linear:**

method 7 of H&F [1]. This method give continuous results using:

- $\alpha = 1$
- $\beta = 1$

**median\_unbiased:**

method 8 of H&F [1]. This method is probably the best method if the sample distribution function is unknown (see reference). This method give continuous results using:

- $\alpha = 1/3$
- $\beta = 1/3$

**normal\_unbiased:**

method 9 of H&F [1]. This method is probably the best method if the sample distribution function is known to be normal. This method give continuous results using:

- $\alpha = 3/8$
- $\beta = 3/8$

**lower:**

NumPy method kept for backwards compatibility. Takes  $i$  as the interpolation point.

**higher:**

NumPy method kept for backwards compatibility. Takes  $j$  as the interpolation point.

**nearest:**

NumPy method kept for backwards compatibility. Takes  $i$  or  $j$ , whichever is nearest.

**midpoint:**

NumPy method kept for backwards compatibility. Uses  $(i + j) / 2$ .

## References

[1]

## 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
>>> 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)

Return 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=<function sum>, label='total', **kwargs) → Array`

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
```

(continues on next page)



(continued from previous page)

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(*axes) → Array`

Return 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,F0')
>>> sex = Axis('sex=M,F')
>>> a = Array([[4, 6], [2, 8]], [nat, sex])
>>> a
nat\sex  M  F
      BE  4  6
      F0  2  8
>>> a.percent()
nat\sex    M    F
      BE 20.0 30.0
      F0 10.0 40.0
>>> a.percent('sex')
nat\sex    M    F
      BE 40.0 60.0
      F0 20.0 80.0
```

## `larray.Array.ratio`

`Array.ratio(*axes) → Array`

Return 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(*axes) → Array`

Return 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 produce a warning message and 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(axis=-1, d=1, label='upper') → Array`

Compute 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 = [[4, 5, 4, 6, 9], [2, 4, 3, 0, 0]]
>>> a = Array(data, "sex=F,M; year=2017..2021")
>>> a
sex\year  2017  2018  2019  2020  2021
      F    4    5    4    6    9
      M    2    4    3    0    0
>>> a.growth_rate()
sex\year  2018  2019  2020  2021
      F  0.25 -0.2  0.5  0.5
      M  1.0 -0.25 -1.0  0.0
>>> a.growth_rate(label='lower')
sex\year  2017  2018  2019  2020
      F  0.25 -0.2  0.5  0.5
      M  1.0 -0.25 -1.0  0.0
>>> a.growth_rate(d=2)
sex\year  2019  2020  2021
      F  0.0  0.2  1.25
      M  0.5 -1.0 -1.0
```

It works on any axis, not just time-based axes

```
>>> a.growth_rate('sex')
sex\year  2017  2018  2019  2020  2021
      M -0.5 -0.2 -0.25 -1.0 -1.0
```

Or part of axes

```
>>> a.growth_rate(a.year[2017:])
sex\year  2018  2019  2020  2021
      F  0.25 -0.2  0.5  0.5
      M  1.0 -0.25 -1.0  0.0
```

## larray.Array.describe

`Array.describe(*args, percentiles=None) → Array`

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(*args, percentiles=None)` → *Array*

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:]))
```

(continues on next page)

(continued from previous page)

```

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
      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

```

## larray.Array.value\_counts

### Array.value\_counts()

Count number of occurrences of each unique value in array.

#### Returns

##### Array of ints

The number of occurrences of each unique value in the input array.

See also:

[\*Array.unique\*](#)

### Examples

```

>>> arr = Array([5, 2, 5, 5, 2, 3, 7], "a=a0..a6")
>>> arr
a  a0  a1  a2  a3  a4  a5  a6
   5   2   5   5   2   3   7
>>> arr.value_counts()
value  2  3  5  7
       2  1  3  1

```

## Sorting

<a href="#"><i>Array.sort_labels</i></a> ([axes, ascending])	Sort labels of axes of the array.
<a href="#"><i>Array.sort_values</i></a> ([key, axis, ascending])	Sort values of the array.
<a href="#"><i>Array.labelsofsorted</i></a> ([axis, ascending, kind])	Return the labels that would sort this array.
<a href="#"><i>Array.indicesofsorted</i></a> ([axis, ascending, kind])	Return the indices that would sort this array.

**larray.Array.sort\_labels**

`Array.sort_labels(axes=None, ascending=True) → Array`

Sort labels of axes of the array.

**Parameters****axes**

[axis reference (Axis, str, int) or list of them, optional] Axes to sort the labels of. Defaults None (all axes).

**ascending**

[bool, optional] Sort labels in ascending order. Defaults to True.

**Returns****Array**

Array with sorted labels.

**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_labels('sex')
nat\sex  F  M
      EU  1  0
      FO  3  2
      BE  5  4
>>> a.sort_labels()
nat\sex  F  M
      BE  5  4
      EU  1  0
      FO  3  2
>>> a.sort_labels(('sex', 'nat'))
nat\sex  F  M
      BE  5  4
      EU  1  0
      FO  3  2
>>> a.sort_labels(ascending=False)
nat\sex  M  F
      FO  2  3
      EU  0  1
      BE  4  5
```

## larray.Array.sort\_values

`Array.sort_values(key=None, axis=None, ascending=True) → Array`

Sort 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
```

(continues on next page)



(continued from previous page)

```

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
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(axis=None, ascending=True, kind='quicksort') → Array`

Return 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(axis=None, ascending=True, kind='quicksort')` → *Array*

Return 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
      0  1  0
      1  0  2
      2  2  1
```

## Reshaping/Extending/Reordering

<code>Array.reshape(target_axes)</code>	Given a list of new axes, changes the shape of the array.
<code>Array.reshape_like(target)</code>	Same as reshape but with an array as input.
<code>Array.compact([display, name])</code>	Detect and remove "useless" axes (ie axes for which values are constant over the whole axis).
<code>Array.reindex([axes_to_reindex, new_axis, ...])</code>	Reorder and/or add new labels in axes.
<code>Array.transpose(*args)</code>	Reorder axes.
<code>Array.expand([target_axes, out, readonly])</code>	Expand this array to target_axes.
<code>Array.prepend(axis, value[, label])</code>	Add an array before this array along an axis.
<code>Array.append(axis, value[, label])</code>	Add a value to this array along an axis.
<code>Array.extend(**kwargs)</code>	
<code>Array.insert(value[, before, after, pos, ...])</code>	Insert value in array along an axis.
<code>Array.broadcast_with(target[, check_compatible])</code>	Return an array that is (NumPy) broadcastable with target.
<code>Array.align(other[, join, fill_value, axes])</code>	Align two arrays on their axes with the specified join method.

## larray.Array.reshape

`Array.reshape(target_axes) → Array`

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
a0     0     1     2     3
a1     4     5     6     7
```

### larray.Array.reshape\_like

`Array.reshape_like(target) → Array`

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(display=False, name='array') → Array`

Detect and remove “useless” axes (ie axes for which values are constant over the whole axis).

### Parameters

#### display

[bool, optional] Whether to display a message with the name of constant axes which were discarded. Defaults to False.

#### name

[str, optional] Name to use in the message if *display* is True. Defaults to “array”.

### 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(axes_to_reindex=None, new_axis=None, fill_value=nan, inplace=False, **kwargs) → Array`

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 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
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
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(*args) → Array`

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
c0 b1   2   6
c1 b0   1   5
c1 b1   3   7
```



## larray.Array.expand

`Array.expand(target_axes=None, out=None, readonly=False) → Array`

Expand this array to `target_axes`.

Target axes will be added to this array if not present. In most cases this function is not needed because LArray can do operations with arrays having different (compatible) axes.

### Parameters

#### `target_axes`

[string, list of Axis or AxisCollection, optional] This array 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) Defaults to False.

### 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 the 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(axis, value, label=None) → Array`

Add an array before this array 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,F0;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
```

(continues on next page)

(continued from previous page)

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(axis, value, label=None) → Array`

Add a value to this array along an axis.

### 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. When *axis* is not present in *value*, this argument should be used. Defaults to None.

### Returns

#### Array

Array with *value* appended along *axis*.

## Examples

```
>>> arr = ones('nat=BE,FO;sex=M,F')
>>> arr["BE", "F"] = 2.0
>>> arr
nat\sex    M    F
    BE  1.0  2.0
    FO  1.0  1.0
>>> sex_total = arr.sum('sex')
>>> sex_total
nat    BE    FO
    3.0  2.0
>>> arr.append('sex', sex_total, label='M+F')
nat\sex    M    F  M+F
    BE  1.0  2.0  3.0
    FO  1.0  1.0  2.0
```

The value can already have the axis along which it is appended:

```
>>> sex_total = arr.sum('sex', keepaxes='M+F')
>>> sex_total
nat\sex  M+F
    BE  3.0
```

(continues on next page)

(continued from previous page)

```

    FO  2.0
>>> arr.append('sex', sex_total)
nat\sex  M    F  M+F
    BE  1.0  2.0  3.0
    FO  1.0  1.0  2.0

```

The value can be a scalar or an array with fewer axes than the original array. In this case, the appended value is expanded (repeated) as necessary:

```

>>> arr.append('nat', 2, 'Other')
nat\sex  M    F
    BE  1.0  2.0
    FO  1.0  1.0
    Other 2.0  2.0

```

The value can also have extra axes (axes not present in the original array), in which case, the original array is expanded as necessary:

```

>>> other = zeros('type=type1,type2')
>>> other
type  type1  type2
    0.0    0.0
>>> arr.append('nat', other, 'Other')
nat  sex\type  type1  type2
    BE      M    1.0    1.0
    BE      F    2.0    2.0
    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(**kwargs)`

## larray.Array.insert

`Array.insert(value, before=None, after=None, pos=None, axis=None, label=None) → Array`

Insert 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
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(target, check_compatible=False) → Array`

Return 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]

**check\_compatible**

[bool, optional] Whether to check that common axes are compatible. Defaults to False.

### Returns

Array

## `larray.Array.align`

`Array.align(other, join='outer', fill_value=nan, axes=None) → Tuple[Array, Array]`

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
```

(continues on next page)

(continued from previous page)

```

>>> aligned2
a\b    b0    b1
a0    0.0   -1.0
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

```

(continues on next page)



(continued from previous page)

```

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')
>>> 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(other[, rtol, atol, ...])</code>	Compare this array with another array and returns True if they have the same axes and elements, False otherwise.
<code>Array.allclose(other[, rtol, atol, ...])</code>	Compare this array with another array and returns True if they are element-wise equal within a tolerance.
<code>Array.eq(other[, rtol, atol, nans_equal])</code>	Compare this array with another array element-wise and returns an array of booleans.
<code>Array.isin(test_values[, assume_unique, invert])</code>	Compute whether each element of this array is in <i>test_values</i> .
<code>Array.nonzero()</code>	Return 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.
<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([axis])</code>	Return labels of the minimum values along a given axis.
<code>Array.indexofmin([axis])</code>	Return indices of the minimum values along a given axis.
<code>Array.labelofmax([axis])</code>	Return labels of the maximum values along a given axis.
<code>Array.indexofmax([axis])</code>	Return indices of the maximum values along a given axis.

## larray.Array.equals

`Array.equals(other, rtol=0, atol=0, nans_equal=False, check_axes=False) → bool`

Compare this array 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 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 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**

Return True if this array is equal to other.

See also:

[\*Array.eq\*](#), [\*Array.allclose\*](#)

### 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.
>>> 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.allclose

`Array.allclose(other: Any, rtol: float = 1e-05, atol: float = 1e-08, nans_equal: bool = True, check_axes: bool = False) → bool`

Compare this array with another array and returns True if they are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference ( $\text{rtol} * \text{abs}(\text{other})$ ) and the absolute difference `atol` are added together to compare against the absolute difference between this array and `other`.

NaN values are treated as equal if they are in the same place and if `nans_equal=True`.

### 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 1e-05.

#### **atol**

[float or int, optional] The absolute tolerance parameter (see Notes). Defaults to 1e-08.

#### **nans\_equal**

[boolean, optional] Whether 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.

#### **check\_axes**

[boolean, optional] Whether 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**

Return True if the two arrays are equal within the given tolerance; False otherwise.

See also:

[`Array.equals`](#)

### Notes

If the following equation is element-wise True, then `allclose` returns True.

$$\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$$

The above equation is not symmetric in `array1` and `array2`, so that `array1.allclose(array2)` might be different from `array2.allclose(array1)` in some rare cases.

## Examples

```
>>> arr1 = Array([1e10, 1e-7], "a=a0,a1")
>>> arr2 = Array([1.00001e10, 1e-8], "a=a0,a1")
>>> arr1.allclose(arr2)
False
```

```
>>> arr1 = Array([1e10, 1e-8], "a=a0,a1")
>>> arr2 = Array([1.00001e10, 1e-9], "a=a0,a1")
>>> arr1.allclose(arr2)
True
```

```
>>> arr1 = Array([1e10, 1e-8], "a=a0,a1")
>>> arr2 = Array([1.0001e10, 1e-9], "a=a0,a1")
>>> arr1.allclose(arr2)
False
```

```
>>> arr1 = Array([1.0, nan], "a=a0,a1")
>>> arr2 = Array([1.0, nan], "a=a0,a1")
>>> arr1.allclose(arr2)
True
>>> arr1.allclose(arr2, nans_equal=False)
False
```

## larray.Array.eq

`Array.eq(other, rtol=0, atol=0, nans_equal=False) → Array`

Compare this array 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 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 this array 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, Array.isclose****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**(*test\_values*, *assume\_unique=False*, *invert=False*) → *Array*

Compute whether each element of this array is in *test\_values*. Return 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()` → `Tuple[IGroup, ...]`

Return 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
```

(continues on next page)



(continued from previous page)

```

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]
>>> 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 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 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
```

(continues on next page)

(continued from previous page)

```
>>> # 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
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 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 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 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 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 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 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'
```

(continues on next page)



(continued from previous page)

```
>>> 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 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 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 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 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
>>> # 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 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 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
>>> # 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 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 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
```

(continues on next page)



(continued from previous page)

```

      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(axis=None) → Union[Array, Tuple[Union[bool, int, float, str, bytes, generic], ...]]`

Return 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(axis=None) → Union[Array, Tuple[int, ...]]`

Return 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(axis=None) → Union[Array, Tuple[Union[bool, int, float, str, bytes, generic], ...]]`

Return 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
>>> arr.labelofmax()
('IT', 'F')
```

## larray.Array.indexofmax

`Array.indexofmax(axis=None) → Union[Array, Tuple[int, ...]]`

Return 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
```

(continues on next page)

(continued from previous page)

```
>>> arr.indexofmax('sex')
nat  BE  FR  IT
      1   0   1
>>> arr.indexofmax()
(2, 1)
```

## Iterating

<code>Array.keys([axes, ascending])</code>	Return a view on the array labels along axes.
<code>Array.values([axes, ascending])</code>	Return a view on the values of the array along axes.
<code>Array.items([axes, ascending])</code>	Return a (label, value) view of the array along axes.

## larray.Array.keys

`Array.keys(axes=None, ascending=True) → Product`

Return 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 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(axes=None, ascending=True) → Union[ndarray, List[Array], ArrayPositionalIndexer]`

Return 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 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
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):
```

(continues on next page)

(continued from previous page)

```
...     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(axes=None, ascending=True) → SequenceZip`

Return 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 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]
>>> 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(other)</code>	Divide this array by other, but return 0.0 where other is 0.
<code>Array.clip([minval, maxval, out])</code>	Clip (limit) the values in an array.
<code>Array.shift(axis[, n])</code>	Shift the cells of the array n-times to the right along axis.
<code>Array.roll([axis, n])</code>	Roll the cells of the array n-times to the right along axis.
<code>Array.diff([axis, d, n, label])</code>	Compute the n-th order discrete difference along a given axis.
<code>Array.unique([axes, sort, sep])</code>	Return unique values (optionally along axes).
<code>Array.to_clipboard(*args, **kwargs)</code>	Send the content of the array to the clipboard.

## larray.Array.divnot0

`Array.divnot0(other) → Array`

Divide this array by other, but return 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.divnot0(b)
nat\sex  M  F
      BE  0.0  1.0
      FO  0.0  3.0
```

Compare this to:

```
>>> a / b
nat\sex    M    F
   BE nan  1.0
   FO inf  3.0
```

## larray.Array.clip

**Array.clip**(*minval=None*, *maxval=None*, *out=None*) → *Array*

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
```

clipping using bounds which vary along an axis

```
>>> lower_bound = Array([-2, 0, 2], 'b=b0..b2')
>>> upper_bound = Array([0, 2, 4], 'b=b0..b2')
>>> arr.clip(lower_bound, upper_bound)
a\b  b0  b1  b2
a0   -2   0   2
a1    0   1   2
a2    0   2   4
```

## larray.Array.shift

`Array.shift(axis, n=1) → Array`

Shift 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(axis=None, n=1) → Array`

Roll 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
    1  2
>>> arr.roll('year', n)
sex\year  2019  2020  2021
      M      2      0      1
      F      4      5      3
```

## larray.Array.diff

**Array.diff**(axis=-1, d=1, n=1, label='upper') → *Array*

Compute 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
```

(continues on next page)

(continued from previous page)

```

      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(axes=None, sort=False, sep='_') → Array`

Return 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 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(*args, **kwargs) → None`

Send the content of the array to the 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([name, dropna])</code>	Convert an Array into a Pandas Series.
<code>Array.to_frame([fold_last_axis_name, dropna])</code>	Convert an Array into a Pandas DataFrame.

### `larray.Array.to_series`

`Array.to_series(name=None, dropna=False) → Series`

Convert 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
    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(fold_last_axis_name=False, dropna=None) → DataFrame`

Convert 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*Plot the data of the array into a graph (window pop-up).

---

### `larray.Array.plot`

**property** `Array.plot`: `PlotObject`

Plot 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  $\geq 2$ )
- 'hexbin' : hexbin plot (if array's dimensions  $\geq 2$ )

##### **ax**

[matplotlib axes object, default None]

##### **subplots**

[boolean, Axis, int, str or tuple, default False] Make several subplots. If True, will make subplots for each combination of labels for all axes except the last. If an Axis, int, str (or tuple of those), it will make subplots for combination of labels of those axes.

##### **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. Defaults to True.

**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)

**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
>>> # let us define an array with some made up data
>>> arr = Array([[5, 20, 5, 10], [6, 16, 8, 11]], 'gender=M,F;year=2018..2021')
```

Simple line plot

```
>>> arr.plot()
>>> # show figure (it also resets it after showing it! Do not call it before_
↪savefig)
>>> plt.show()
```

Line plot with grid and a title

```
>>> arr.plot(grid=True, title='line plot')
>>> # save figure in a file (see matplotlib.pyplot.savefig documentation for more_
↪details)
>>> plt.savefig('my_file.png')
```

2 bar plots (one for each gender) sharing the same y axis, which makes sub plots easier to compare. By default sub plots are independant of each other and the axes ranges are computed to “fit” just the data for their individual plot.

```
>>> arr.plot.bar(subplots='gender', sharey=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=(10, 8), tight_layout=True)
>>> # line plot with 2 curves (Males and Females) in the top left corner (0, 0)
>>> arr.plot(ax=ax[0, 0], title='line plot')
>>> # bar plot with stacked values in the top right corner (0, 1)
>>> arr.plot.bar(ax=ax[0, 1], stacked=True, title='stacked bar plot')
>>> # area plot in the bottom left corner (1, 0)
>>> arr.plot.area(ax=ax[1, 0], title='area plot')
>>> # scatter plot in the bottom right corner (1, 1), using the year as color
>>> # index and a specific colormap
>>> arr.plot.scatter(ax=ax[1, 1], x='M', y='F', c=arr.year, colormap='viridis',
...                  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 for monotonically increasing sample points.
<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>isscalar(element)</code>	Return <i>True</i> if the type of element is a scalar type.
<code>isnans(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 normalized 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
```

(continues on next page)

(continued from previous page)

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
```

(continues on next page)



(continued from previous page)

```
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.

See also:

**scipy.linalg.inv**  
Similar function in SciPy.

## 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 for monotonically increasing sample points.

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

**Warning:** 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)
```

See also:

`scipy.interpolate`

**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()
```

The *abs* function can be used as a shorthand for `np.absolute` on ndarrays.

```
>>> x = np.array([-1.2, 1.2])
>>> abs(x)
array([1.2, 1.2])
```

## 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.isscalar**

`larray.isscalar(element: Any) → bool`

Return *True* if the type of *element* is a scalar type.

**Parameters**

**element: any**

Input argument, can be of any type and shape.

**Returns**

**bool**

*True* if *element* is a scalar type, *False* if it is not.

**Examples**

```
>>> from larray import ndtest
>>> isscalar(3.1)
True
>>> isscalar([3.1])
False
>>> isscalar(False)
True
>>> isscalar('larray')
True
```

```
>>> arr = ndtest((2, 2))
>>> arr
a\b  b0  b1
a0    0   1
a1    2   3
>>> isscalar(arr)
False
```



## 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=3333333, neginf=3333333)
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,
```

(continues on next page)

(continued from previous page)

```
0.000000000e+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:

### emath.sqrt

A version which returns complex numbers when given negative reals. Note that 0.0 and -0.0 are handled differently for complex inputs.

## Notes

*sqrt* has—consistent with common convention—as its branch cut the real “interval”  $[-\infty, 0)$ , 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  $I_0$ .

### Parameters

**x**

[array\_like of float] Argument of the Bessel function.

### Returns

**out**

[ndarray, shape = x.shape, dtype = float] The modified Bessel function evaluated at each of the elements of *x*.

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  $(8,\infty)$ , 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  $5.8\text{e-}16$  with an rms of  $1.4\text{e-}16$  ( $n = 30000$ ).

## References

[1], [2], [3]

## Examples

```
>>> np.i0(0.)
array(1.0)
>>> np.i0([0, 1, 2, 3])
array([1.          , 1.26606588, 2.2795853 , 4.88079259])
```

## larray.sinc

`larray.sinc(*args, **kwargs)`

Return the normalized sinc function.

larray specific variant of `numpy.sinc`.

Documentation from numpy:

The sinc function is equal to  $\sin(\pi x)/(\pi x)$  for any argument  $x \neq 0$ . `sinc(0)` takes the limit value 1, making `sinc` not only everywhere continuous but also infinitely differentiable.

---

**Note:** Note the normalization factor of `pi` used in the definition. This is the most commonly used definition in signal processing. Use `sinc(x / np.pi)` to obtain the unnormalized sinc function  $\sin(x)/x$  that is more common in mathematics.

---

### 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

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>	Evenly round 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)`

Evenly round to the given number of decimals.

larray specific variant of `numpy.round`.

Documentation from numpy:

### Parameters

**a**

[array\_like] Input data.

**decimals**

[int, optional] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary. See `ufuncs-output-type` for more details.

### Returns

**rounded\_array**

[ndarray] An array of the same type as *a*, containing the rounded values. Unless *out* was specified, a new array is created. A reference to the result is returned.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float.

See also:

**ndarray.round**

equivalent method

**around**

an alias for this function

*ceil*, *fix*, *floor*, *rint*, *trunc*

### Notes

For values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc.

`np.round` uses a fast but sometimes inexact algorithm to round floating-point datatypes. For positive *decimals* it is equivalent to `np.true_divide(np.rint(a * 10**decimals), 10**decimals)`, which has error due to the inexact representation of decimal fractions in the IEEE floating point standard [1] and errors introduced when scaling by powers of ten. For instance, note the extra “1” in the following:

```
>>> np.round(56294995342131.5, 3)
56294995342131.51
```

If your goal is to print such values with a fixed number of decimals, it is preferable to use numpy’s float printing routines to limit the number of printed decimals:

```
>>> np.format_float_positional(56294995342131.5, precision=3)
'56294995342131.5'
```

The float printing routines use an accurate but much more computationally demanding algorithm to compute the number of digits after the decimal point.

Alternatively, Python's builtin *round* function uses a more accurate but slower algorithm for 64-bit floating point values:

```
>>> round(56294995342131.5, 3)
56294995342131.5
>>> np.round(16.055, 2), round(16.055, 2) # equals 16.054999999999997
(16.06, 16.05)
```

## References

[1]

## Examples

```
>>> np.round([0.37, 1.64])
array([0., 2.])
>>> np.round([0.37, 1.64], decimals=1)
array([0.4, 1.6])
>>> np.round([.5, 1.5, 2.5, 3.5, 4.5]) # rounds to nearest even value
array([0., 2., 2., 4., 4.])
>>> np.round([1,2,3,11], decimals=1) # ndarray of ints is returned
array([ 1,  2,  3, 11])
>>> np.round([1,2,3,11], decimals=-1)
array([ 0,  0,  0, 10])
```

## 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\*](#), [\*fix\*](#)

**Notes**

Some spreadsheet programs calculate the “floor-towards-zero”, where `floor(-2.5) == -2`. NumPy instead uses the definition of *floor* where `floor(-2.5) == -3`. The “floor-towards-zero” function is called `fix` in NumPy.

**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*, *fix*

**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*, *fix***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:

*fix*, *ceil*, *floor*, *trunc*

## Notes

For values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc.

## 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

**out**

[ndarray, optional] A location into which the result is stored. If provided, it must have a shape that the input broadcasts to. If not provided or None, a freshly-allocated array is returned.

### Returns

**out**

[ndarray of floats] A float array with the same dimensions as the input. If second argument is not supplied then a float array is returned with the rounded values.

If a second argument is supplied the result is stored there. The return value *out* is then a reference to that array.

See also:

[\*rint\*](#), [\*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](#), [math.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  $[-inf, 0]$  and is continuous from above on it. *log* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

In the cases where the input has a negative real part and a very small negative complex part (approaching 0), the result is so close to  $-pi$  that it evaluates to exactly  $-pi$ .

## 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  $[-inf, 0]$  and is continuous from above on it. *log10* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

In the cases where the input has a negative real part and a very small negative complex part (approaching 0), the result is so close to  $-pi$  that it evaluates to exactly  $-pi$ .

## 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\*](#), [`math.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  $[-inf, 0]$  and is continuous from above on it. *log2* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

In the cases where the input has a negative real part and a very small negative complex part (approaching 0), the result is so close to  $-pi$  that it evaluates to exactly  $-pi$ .

## 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 <code>x1/x2</code> 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 taking the complement of large deltas with respect to the period.

## 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 ( $2\pi$  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  $\pi$ . Sines of angles between  $\pi$  and  $2\pi$  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)
```

(continues on next page)

(continued from previous page)

```
>>> 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\*](#), [\*emath.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. [https://personal.math.ubc.ca/~cbm/aands/page\\_79.htm](https://personal.math.ubc.ca/~cbm/aands/page_79.htm)

## 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\*](#), `math.accos`

### 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  $[-\infty, -1]$  and  $[1, \infty]$  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. [https://personal.math.ubc.ca/~cbm/aands/page\\_79.htm](https://personal.math.ubc.ca/~cbm/aands/page_79.htm)

### 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, infj]` and `[-1j, -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. [https://personal.math.ubc.ca/~cbm/aands/page\\_79.htm](https://personal.math.ubc.ca/~cbm/aands/page_79.htm)

## 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 cast-able 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 taking the complement of large deltas with respect to the period.

larray specific variant of `numpy.unwrap`.

Documentation from numpy:

This unwraps a signal  $p$  by changing elements which have an absolute difference from their predecessor of more than `max(discont, period/2)` to their *period*-complementary values.

For the default case where *period* is  $2\pi$  and *discont* is  $\pi$ , this unwraps a radian phase  $p$  such that adjacent differences are never greater than  $\pi$  by adding  $2k\pi$  for some integer  $k$ .

### Parameters

**p**

[array\_like] Input array.

**discont**

[float, optional] Maximum discontinuity between values, default is `period/2`. Values below `period/2` are treated as if they were `period/2`. To have an effect different from the default, *discont* should be larger than `period/2`.

**axis**

[int, optional] Axis along which unwrap will operate, default is the last axis.

**period**

[float, optional] Size of the range over which the input wraps. By default, it is `2 * pi`.

New in version 1.21.0.

### Returns

**out**

[ndarray] Output array.

See also:

`rad2deg`, `deg2rad`

## Notes

If the discontinuity in  $p$  is smaller than  $\text{period}/2$ , but larger than  $\text{discont}$ , no unwrapping is done because taking the 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
>>> np.unwrap([0, 1, 2, -1, 0], period=4)
array([0, 1, 2, 3, 4])
>>> np.unwrap([ 1, 2, 3, 4, 5, 6, 1, 2, 3], period=6)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.unwrap([2, 3, 4, 5, 2, 3, 4, 5], period=4)
array([2, 3, 4, 5, 6, 7, 8, 9])
>>> phase_deg = np.mod(np.linspace(0, 720, 19), 360) - 180
>>> np.unwrap(phase_deg, period=360)
array([-180., -140., -100.,  -60.,  -20.,   20.,   60.,  100.,  140.,
        180.,  220.,  260.,  300.,  340.,  380.,  420.,  460.,  500.,
        540.])
```

## Hyperbolic functions

<a href="#"><code>sinh</code></a> ( <i>x</i> , /[, out, where, casting, order, ...])	Hyperbolic sine, element-wise.
<a href="#"><code>cosh</code></a> ( <i>x</i> , /[, out, where, casting, order, ...])	Hyperbolic cosine, element-wise.
<a href="#"><code>tanh</code></a> ( <i>x</i> , /[, out, where, casting, order, ...])	Compute hyperbolic tangent element-wise.
<a href="#"><code>arcsinh</code></a> ( <i>x</i> , /[, out, where, casting, order, ...])	Inverse hyperbolic sine element-wise.
<a href="#"><code>arccosh</code></a> ( <i>x</i> , /[, out, where, casting, order, ...])	Inverse hyperbolic cosine, element-wise.
<a href="#"><code>arctanh</code></a> ( <i>x</i> , /[, out, where, casting, order, ...])	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  $1/2 * (\text{np.exp}(x) + \text{np.exp}(-x))$  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)
```

(continues on next page)

(continued from previous page)

```
>>> 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  $[-1, -inf]$  and  $[1, inf]$  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

<code>angle(*args, **kwargs)</code>	Return the angle of the complex argument.
<code>real(*args, **kwargs)</code>	Return the real part of the complex argument.
<code>imag(*args, **kwargs)</code>	Return the imaginary part of the complex argument.
<code>conj(x, /[, out, where, casting, order, ...])</code>	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`.

Changed in version 1.16.0: This function works on subclasses of ndarray like *ma.array*.

See also:

[\*arctan2\*](#)  
[\*absolute\*](#)

**Notes**

Although the angle of the complex number 0 is undefined, `numpy.angle(0)` returns the value 0.

**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

<code>signbit(x, /[, out, where, casting, order, ...])</code>	Returns element-wise True where signbit is set (less than zero).
<code>copysign(x1, x2, /[, out, where, casting, ...])</code>	Change the sign of x1 to that of x2, element-wise.
<code>frexp(x[, out1, out2], / [[, out, where, ...])</code>	Decompose the elements of x into mantissa and twos exponent.
<code>ldexp(x1, x2, /[, out, where, casting, ...])</code>	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 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

```
>>> 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__(*args, **kwargs)
```

## Methods

<code>__init__(*args, **kwargs)</code>	
<code>clear()</code>	
<code>copy()</code>	
<code>from_array(array)</code>	
<code>from_hdf(hdfstore[, key])</code>	
<code>fromkeys(iterable[, value])</code>	Create a new dictionary with keys from iterable and values set to value.
<code>get(key[, default])</code>	Return the value for key if key is in the dictionary, else default.
<code>items()</code>	
<code>keys()</code>	
<code>pop(key[, default])</code>	If the key is not found, return the default if given; otherwise, raise a <code>KeyError</code> .
<code>popitem(/)</code>	Remove and return a (key, value) pair as a 2-tuple.
<code>setdefault(key[, default])</code>	Insert key with a value of default if key is not in the dictionary.
<code>to_hdf(hdfstore[, key])</code>	
<code>update([E, ]**F)</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>	Read 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>	Read excel file from sheet name and returns an Array with the contents.
<code>read_hdf(filepath_or_buffer, key[, ...])</code>	Read a scalar or an axis or group or array named key from a HDF5 file in filepath (path+name).
<code>read_eurostat(filepath_or_buffer, **kwargs)</code>	Read EUROSTAT TSV (tab-separated) file into an array.
<code>read_sas(filepath[, nb_axes, index_col, ...])</code>	Read sas file and returns an Array with the contents
<code>read_stata(filepath_or_buffer[, index_col, ...])</code>	Read 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, decimal='.', fill_value=nan, na=nan, sort_rows=False, sort_columns=False, wide=True, dialect='larray', **kwargs) → Array`

Read 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 to use. Defaults to ','.

##### **headersep**

[str or None, optional] Specific separator to use for headers. Defaults to None (uses `sep`).

##### **decimal**

[str, optional] Character to use as decimal point. Defaults to '.'.

##### **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 to sort the rows alphabetically (sorting is more efficient than not sorting). Defaults to False.

**sort\_columns**

[bool, optional] Whether to sort the columns alphabetically (sorting is more efficient than not sorting). Defaults to False.

**wide**

[bool, optional] Whether 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 / 'population.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 / 'population_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
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 use 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 / 'population_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
```

(continues on next page)

(continued from previous page)

```

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 / 'population_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())
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) → Array`

## 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) → Array`

Read excel file from sheet name and returns an Array with the contents.

### Parameters

#### filepath

[str or Path] 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 to sort the rows alphabetically (sorting is more efficient than not sorting). Defaults to False.

**sort\_columns**

[bool, optional] Whether to sort the columns alphabetically (sorting is more efficient than not sorting). Defaults to False.

**wide**

[bool, optional] Whether 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**

[['xlwings', 'openpyxl', 'xlrd'], optional] Engine to use to read the Excel file. The 'xlrd' engine must be used to read Excel files with the old '.xls' extension. Either 'xlwings' or 'openpyxl' can be used to read Excel files with the standard '.xlsx' extension. Defaults to 'xlwings' if the module is installed, 'openpyxl' 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 ‘population\_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='population_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='population_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 'population_missing_axis_name' as is
>>> arr = read_excel(fname, sheet='population_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='population_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 ‘population\_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, 'population_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, 'population_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) → Array`

Read a scalar or an axis or group or array named key from a HDF5 file in filepath (path+name).

### Parameters

#### filepath\_or\_buffer

[str or Path or pandas.HDFStore] Path and name where the HDF5 file is stored or a HDF-Store object.

#### key

[str or Group] Name of the scalar or axis or group or 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 to sort the rows alphabetically. Must be False if the read array has been dumped with an larray version  $\geq 0.30$ . Defaults to False.

#### sort\_columns

[bool, optional] Whether to sort the columns alphabetically. Must be False if the read array has been dumped with an larray version  $\geq 0.30$ . 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) → Array`

Read 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) → Array`

### Read 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) → Array`

Read 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 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 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`) loses 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

<code>Array.to_csv(filepath[, sep, na_rep, wide, ...])</code>	Write array to a csv file.
<code>Array.to_excel([filepath, sheet, position, ...])</code>	Write array in the specified sheet of specified excel workbook.
<code>Array.to_hdf(filepath, key)</code>	Write array to a HDF file.
<code>Array.to_stata(filepath_or_buffer, **kwargs)</code>	Write array to a Stata .dta file.
<code>Array.dump(self[, header, wide, value_name, ...])</code>	Dump array as a 2D nested list.

### larray.Array.to\_csv

`Array.to_csv(filepath, sep=',', na_rep='', wide=True, value_name='value', dropna=None, dialect='default', **kwargs) → None`

Write array to a csv file.

#### Parameters

##### filepath

[str or Path] 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 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 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

```
>>> tmp_path = getfixture('tmp_path')
>>> fname = tmp_path / 'test.csv'
>>> a = ndtest('nat=BE,F0;sex=M,F')
>>> a
nat\sex  M  F
      BE  0  1
      F0  2  3
>>> a.to_csv(fname)
>>> with open(fname) as f:
```

(continues on next page)

(continued from previous page)

```

...     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**(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) → None

Write array in the specified sheet of specified excel workbook.

### Parameters

#### filepath

[str or Path 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 to overwrite the existing file (or just modify the specified sheet).

Defaults to False.

**clear\_sheet**

[bool, optional] Whether to clear the existing sheet (if any) before writing. Defaults to False.

**header**

[bool, optional] Whether to write a header (axes names and labels). Defaults to True.

**transpose**

[bool, optional] Whether to transpose the array over last axis. This is equivalent to paste with option transpose in Excel. Defaults to False.

**wide**

[boolean, optional] Whether 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,F0;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(filepath, key) → None`

Write 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 or Path] 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(filepath_or_buffer, **kwargs) → None`

Write 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`) loose 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  country  sex
          0         BE    F
          1         FR    M
          2         FR    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 to output axes names and labels.

**wide**

[boolean, optional] Whether 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 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 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 of builtin Python values 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],
```

(continues on next page)

(continued from previous page)

```

['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

---

<code>open_excel</code> ([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, str or Path, 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 to overwrite an existing file, if any. Defaults to False.



**visible**

[None or bool, optional] whether 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 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 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()</code>	Return the names of the Excel sheets.
<code>Workbook.save([path])</code>	Saves the Workbook.
<code>Workbook.close()</code>	Close the workbook in Excel.
<code>Workbook.app()</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__`(*filepath=None, overwrite\_file=False, visible=None, silent=None, app=None, load\_addins=None*)

### Methods

<code>__init__</code> ( <i>filepath, overwrite_file, ...</i> )	
<code>app</code> ()	Return the Excel instance this workbook is attached to.
<code>close</code> ()	Close the workbook in Excel.
<code>save</code> ( <i>[path]</i> )	Saves the Workbook.
<code>sheet_names</code> ()	Return the names of the Excel sheets.

## larray.Workbook.sheet\_names

`Workbook.sheet_names()`

Return 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(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 or Path, 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()`

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()`

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 '.crtx' extension).
<code>ExcelReport.template</code>	Set a default Excel template file.
<code>ExcelReport.set_item_default_size(kind[, ...])</code>	Override the default 'width' and 'height' values for the given kind of item.
<code>ExcelReport.graphs_per_row</code>	Default number of graphs per row.
<code>ExcelReport.new_sheet(sheet_name)</code>	Add a new empty output sheet.
<code>ExcelReport.sheet_names()</code>	Return the names of the output sheets.
<code>ExcelReport.to_excel(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 or Path, optional] Path to the directory containing the Excel template files (with a '.crtx' extension). Defaults to None.

###### template

[str or Path, 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.population['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.population['Belgium', 'Female'], 'Population - Female')
>>> sheet_be.add_graph(demo.population['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.population.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 in {year}': ↪
↪ population},
...                           {'gender': population.gender, 'year': population.
↪ time},
...                           template='line', width=450, height=250, graphs_per_
↪ row=2)
```

Generate the report Excel file

```
>>> report.to_excel('Demography_Report.xlsx')
```

```
__init__()
```

## Methods

---

<code>__init__()</code>	
<code>new_sheet(sheet_name)</code>	Add a new empty output sheet.
<code>set_item_default_size(kind[, width, height])</code>	Override the default 'width' and 'height' values for the given kind of item.
<code>sheet_names()</code>	Return the names of the output sheets.
<code>to_excel(filepath[, data_sheet_name, overwrite])</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 '.crtx' extension).

---

### `larray.ExcelReport.template_dir`

#### property `ExcelReport.template_dir`

Set the path to the directory containing the Excel template files (with '.crtx' 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 or Path] 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`

[str or Path] 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.

## 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_population = report.new_sheet('Population')
>>> sheet_population.add_graph(demo.population['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_population.template = r'C:\other_templates_dir\Line_Marker.crtx'
>>> sheet_population.add_graph(demo.population['Germany'], 'Germany')
```

## `larray.ExcelReport.set_item_default_size`

`ExcelReport.set_item_default_size(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(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:** `ReportSheet`

## 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.population['Belgium'], 'Population', template='Line')
```



## larray.ExcelReport.sheet\_names

ExcelReport.sheet\_names()

Return the names of the output sheets.

### Examples

```
>>> report = ExcelReport()
>>> sheet_population = 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(filepath, data\_sheet\_name='\_\_data\_\_', overwrite=True)

Generate the report Excel file.

### Parameters

#### filepath

[str or Path] 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 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.population[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

<i>ReportSheet()</i>	Represents a sheet dedicated to contains only graphical items (title banners, graphs).
<i>ReportSheet.template_dir</i>	Set the path to the directory containing the Excel template files (with '.crtx' extension).
<i>ReportSheet.template</i>	Set a default Excel template file.
<i>ReportSheet.set_item_default_size(kind[, ...])</i>	Override the default 'width' and 'height' values for the given kind of item.
<i>ReportSheet.graphs_per_row</i>	Default number of graphs per row.
<i>ReportSheet.add_title(title[, width, ...])</i>	Add a title item to the current sheet.
<i>ReportSheet.add_graph(data[, title, ...])</i>	Add a graph item to the current sheet.
<i>ReportSheet.add_graphs(array_per_title, ...)</i>	Add multiple graph items to the current sheet.
<i>ReportSheet.newline()</i>	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 or Path, optional] Path to the directory containing the Excel template files (with a '.crtx' extension). Defaults to None.

###### template

[str or Path, 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__()`

## Methods

<code>__init__()</code>	
<code>add_graph(data[, title, template, width, ...])</code>	Add a graph item to the current sheet.
<code>add_graphs(array_per_title, ...[, template, ...])</code>	Add multiple graph items to the current sheet.
<code>add_title(title[, width, height, fontsize])</code>	Add a title item to the current sheet.
<code>newline()</code>	Force a new row of graphs.
<code>set_item_default_size(kind[, width, height])</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 '.crtx' extension).

### `larray.ReportSheet.template_dir`

#### property `ReportSheet.template_dir`

Set the path to the directory containing the Excel template files (with '.crtx' 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 or Path] 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**

[str or Path] 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.

## **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_population = report.new_sheet('Population')
>>> sheet_population.add_graph(demo.population['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_population.template = r'C:\other_templates_dir\Line_Marker.crtx'
>>> sheet_population.add_graph(demo.population['Germany'], 'Germany')
```

## **larray.ReportSheet.set\_item\_default\_size**

ReportSheet.set\_item\_default\_size(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(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(data, title=None, template=None, width=None, height=None, min_y=None, max_y=None, xticks_spacing=None, customize_func=None, customize_kwargs=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 or Path, 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.

#### **min\_y: int, optional**

minimum value for the Y axis.

#### **max\_y: int, optional**

maximum value for the Y axis.

#### **xticks\_spacing: int, optional**

space interval between two ticks along the X axis.

#### **customize\_func: function, optional**

user defined function to personalize the graph. The function must take the Chart object as first argument. All keyword arguments defined in `customize_kwargs` are passed to the function at call.

#### **customize\_kwargs: dict, optional**

keywords arguments passed to the function `customize_func` at call.

## 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.population['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')
```

Specify the minimum and maximum values for the Y axis

```
>>> sheet_be.add_graph(demo.population['Belgium'],
...                    'Population (min/max Y axis = 5/6 millions)',
...                    min_y=5e6, max_y=6e6)
```

Specify the interval between two ticks (X axis)

```
>>> sheet_be.add_graph(demo.population['Belgium'], 'Population (every 2 years)',
↳ xticks_spacing=2)
```

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(array_per_title, axis_per_loop_variable, template=None, width=None, height=None, graphs_per_row=1, min_y=None, max_y=None, xticks_spacing=None, customize_func=None, customize_kwargs=None)`

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 or Path, 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.

**min\_y: int, optional**

minimum value for the Y axis.

**max\_y: int, optional**

maximum value for the Y axis.

**xticks\_spacing: int, optional**

space interval between two ticks along the X axis.

**customize\_func: function, optional**

user defined function to personalize the graph. The function must take the Chart object as first argument. All keyword arguments defined in `customize_kwargs` are passed to the function at call.

**customize\_kwargs: dict, optional**

keywords arguments passed to the function `customize_func` at call.

## Examples

```
>>> demo = load_example_data('demography_eurostat')
>>> report = ExcelReport(EXAMPLE_EXCEL_TEMPLATES_DIR)
```

```
>>> sheet_population = report.new_sheet('Population')
>>> population = demo.population
```

Generate a new graph for each combination of gender and year

```
>>> sheet_population.add_graphs(
...     {'Population of {gender} by country in {year}': population},
...     {'gender': population.gender, 'year': population.time},
...     template='line', width=450, height=250, graphs_per_row=2)
```

Specify the minimum and maximum values for the Y axis

```
>>> sheet_population.add_graphs({'Population of {gender} by country for the year
↳ {year}': population},
...                             {'gender': population.gender, 'year': population.time},
...                             template='line', width=450, height=250, graphs_per_row=2,
↳ min_y=0, max_y=50e6)
```



Specify the interval between two ticks (X axis)

```
>>> sheet_population.add_graphs({'Population of {gender} by country for the year
↳ {year}': population},
...                               {'gender': population.gender, 'year': population.time},
...                               template='line', width=450, height=250, graphs_per_row=2,
↳ xticks_spacing=2)
```

```
>>> # do not forget to call 'to_excel' to create the report file
>>> report.to_excel('Demography_Report.xlsx')
```

## larray.ReportSheet.newline

ReportSheet.newline()

Force a new row of graphs.

## 4.3.12 Miscellaneous

<code>asarray(a[, meta])</code>	Convert input as Array if possible.
<code>from_frame(df[, sort_rows, sort_columns, ...])</code>	Convert Pandas DataFrame into Array.
<code>from_series(s[, sort_rows, fill_value, meta])</code>	Convert 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>	Return an array with specified axes and the combination of corresponding labels as values.
<code>union(*args)</code>	Return the union of several "value strings" as a list.
<code>stack([elements, axes, title, meta, dtype, ...])</code>	Combine several arrays or sessions along an axis.
<code>identity(axis)</code>	
<code>diag(a[, k, axes, ndim, split])</code>	Extract a diagonal or construct a diagonal array.
<code>eye(rows[, columns, k, title, dtype, meta])</code>	Return 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[, doc])</code>	Wrap a function using numpy arrays to work with LArray arrays instead.
<code>zip_array_values(values[, axes, ascending])</code>	Return a sequence as if simultaneously iterating on several arrays.
<code>zip_array_items(values[, axes, ascending])</code>	Return a sequence as if simultaneously iterating on several arrays as well as the current iteration "key".

## larray.asarray

`larray.asarray(a, meta=None) → Array`

Convert input as Array if possible.

### Parameters

**a**

[array-like] Input array to convert into an Array.

**meta**

[list of pairs or dict 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) → Array`

Convert 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 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 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 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 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 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 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    a0  b0    0   1
a0    a0  b1    2   3
a1    a1  b0    4   5
a1    a1  b1    6   7
```

## **larray.from\_series**

**larray.from\_series**(*s*, *sort\_rows=False*, *fill\_value=nan*, *meta=None*, *\*\*kwargs*) → *Array*

Convert Pandas Series into Array.

### **Parameters**

**s**

[Pandas Series] Input Pandas Series.

**sort\_rows**

[bool, optional] Whether 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 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
```

(continues on next page)

(continued from previous page)

```

      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) → Path`

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
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__(**kwargs)
```

## Methods

---

```
__init__(**kwargs)
```

---

### `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) → Array`

Return 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 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) → List[Any]`

Return 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) → Array`

Combine several arrays or sessions along an axis.

### Parameters

#### **elements**

[tuple, list, dict or Session.] Elements to stack. Elements can be scalars, arrays, sessions, (label, value) pairs or a {label: value} mapping.

Stacking a single session will stack all its arrays in a single array. Stacking several sessions will take the corresponding arrays in all the sessions and stack them, returning a new session. 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 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 or Session**

A single Array combining input values, or a single Session combining input Sessions. 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
```

(continues on next page)

(continued from previous page)

```

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)

```

>>> 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,
...      ('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) → Array`

Extract 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 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) → Array`

Return a 2-D array with ones on the diagonal and zeros elsewhere.

### Parameters

#### rows

[int or Axis or tuple or length 2 AxisCollection] Rows of the output (if int or Axis) or rows and columns (if tuple or AxisCollection).

#### columns

[int or Axis, optional] Columns of the output. Defaults to the value of *rows* if it is an int or Axis.

#### 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 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('sex=M,F')
sex\sex    M    F
      M  1.0  0.0
      F  0.0  1.0
>>> eye(2, dtype=int)
{0}*{1}*    0    1
           0    1    0
           1    0    1
>>> age = Axis('age=0..2')
>>> sex = Axis('sex=M,F')
>>> 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 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
```

(continues on next page)

(continued from previous page)

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, doc=None)`

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.

##### **doc**

[str, optional] The documentation (docstring) for the new function. Defaults to the documentation of the original function, if any.

#### Returns

##### **function**

A function taking `larray.Array` 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) → SequenceZip`

Return a sequence as if simultaneously iterating on several arrays.

### Parameters

#### values

[sequence of (scalar or Array)] Values to iterate on. Scalars are repeated as many times as necessary.

#### axes

[int, str or Axis or tuple of them, optional] Axis or axes along which to iterate and in which order. All those axes must be compatible (if present) between the different values. Defaults to None (union of all axes present in all arrays, in the order they are found).

#### ascending

[bool, optional] Whether 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
```

When the axis to iterate on (*c* in this case) is not present in one of the arrays (*arr1*), that array is repeated for each label of that axis:

```
>>> 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
a  a0  a1
   1   3
```

When no *axes* are given, it iterates on the union of all compatible axes (*a*, *b*, and *c* in this case):

```
>>> for a1, a2 in zip_array_values((arr1, arr2)):
...     print(f"arr1: {a1}, arr2: {a2}")
arr1: 0, arr2: 0
arr1: 0, arr2: 1
```

(continues on next page)

(continued from previous page)

```

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) → SequenceZip`

Return 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 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[, meta])</code>	Groups several objects together.
<code>arrays([depth, include_private, meta])</code>	Return 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>	Return a session containing all local arrays sorted in alphabetical order.
<code>global_arrays([depth, include_private, meta])</code>	Return 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, meta=None, \*\*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 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

```
>>> # scalars
>>> i, s = 5, 'string'
>>> # 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)

```
>>> ses = Session([(i, i), (s, s), (a, a), (b, b), (a01, a01),
...               (arr1, arr1), (arr2, arr2)])
```

create a Session using keyword arguments

```
>>> ses = Session(i=i, s=s, a=a, b=b, a01=a01, arr1=arr1, arr2=arr2)
```

create a Session by passing a dictionary

```
>>> ses = Session({'i': i, 's': s, 'a': a, 'b': b, 'a01': a01, 'arr1': arr1, 'arr2':
↪ arr2})
```

load Session from file

```
>>> ses = Session('my_session.h5')
```

create a session with metadata

```
>>> ses = Session(arr1=arr1, arr2=arr2, meta=Metadata(title='my title', author=
↪ 'John Smith'))
>>> ses.meta
title: my title
author: John Smith
```

```
__init__(*args, meta=None, **kwargs)
```

## Methods

<code>__init__(*args[, meta])</code>	
<code>add(*args, **kwargs)</code>	Deprecated.
<code>apply(func, *args[, kind])</code>	Apply function <i>func</i> on elements of the session and return a new session.
<code>array_equals(**kwargs)</code>	
<code>compact([display])</code>	Detect and remove "useless" axes (ie axes for which values are constant over the whole axis) for all array objects in session.
<code>copy()</code>	Return a copy of the session.
<code>dump(**kwargs)</code>	
<code>dump_csv(**kwargs)</code>	
<code>dump_excel(**kwargs)</code>	
<code>dump_hdf(**kwargs)</code>	
<code>element_equals(other[, rtol, atol, nans_equal])</code>	Test if each element (group, axis and array) of the current session equals the corresponding element of another session.
<code>equals(other[, rtol, atol, nans_equal])</code>	Test if all elements (groups, axes and arrays) of the current session are equal to those of another session.
<code>filter([pattern, kind])</code>	Return a new session with objects which match some criteria.
<code>get(key[, default])</code>	Return the object corresponding to the key.
<code>items()</code>	Return a view of the session's items ((key, value) pairs).
<code>keys()</code>	Return a view on the session's keys.
<code>load(fname[, names, engine, display])</code>	Load objects from a file, or several .csv files.
<code>save(fname[, names, engine, overwrite, display])</code>	Dump objects from the current session to a file, or several .csv files.
<code>summary([template])</code>	Return a summary of the content of the session.
<code>to_csv(fname[, names, display])</code>	Dump Array objects from the current session to CSV files.
<code>to_excel(fname[, names, overwrite, display])</code>	Dump Array objects from the current session to an Excel file.
<code>to_globals([names, depth, warn, inplace])</code>	Create global variables out of objects in the session.
<code>to_hdf(fname[, names, overwrite, display])</code>	Dump objects from the current session to an HDF file.
<code>to_pickle(fname[, names, overwrite, display])</code>	Dump objects from the current session to a file using pickle.
<code>transpose(*args)</code>	Reorder axes of arrays in session, ignoring missing axes for each array.
<code>update([other])</code>	Update the session with the key/value pairs from other or passed keyword arguments, overwriting existing keys.
<code>values()</code>	Return a view on the session's values.

## Attributes

<code>memory_used</code>	Return the memory consumed by the session in human readable form.
<code>meta</code>	Return metadata of the session.
<code>names</code>	Return the list of names of the objects in the session.
<code>nbytes</code>	Return the memory in bytes consumed by the session.

## `larray.arrays`

`larray.arrays(depth=0, include_private=False, meta=None) → Session`

Return 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 to include private arrays (i.e. arrays starting with `_`). Defaults to False.

#### **meta**

[list of pairs or dict 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) → Session`

Return 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 to include private local arrays (i.e. arrays starting with `_`). Defaults to False.

#### **meta**

[list of pairs or dict 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) → Session`

Return 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 to include private globals arrays (i.e. arrays starting with `_`). Defaults to False.

#### **meta**

[list of pairs or dict 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)
```

(continues on next page)



(continued from previous page)

```

citizenship: citizenship ['Belgium' 'Luxembourg' 'Netherlands'] (3)
time: time [2013 2014 2015 2016 2017] (5)
even_years: time[2014 2016] >> even_years (2)
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>	Return the list of names of the objects in the session.
<code>Session.keys()</code>	Return a view on the session's keys.
<code>Session.values()</code>	Return a view on the session's values.
<code>Session.items()</code>	Return a view of the session's items ((key, value) pairs).
<code>Session.summary([template])</code>	Return a summary of the content of the session.

## larray.Session.names

**property** `Session.names`: `List[str]`

Return 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**() → [KeysView](#)[str]

Return 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**() → [ValuesView](#)[Any]

Return 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()` → `ItemsView[str, Any]`

Return 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(f"{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(template=None)` → `str`

Return 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 f"{key} -> {axes_names} ({shape})\n  title = {array.meta.title}\n  dtype = {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()</code>	Return a copy of the session.
-----------------------------	-------------------------------

---

### `larray.Session.copy`

`Session.copy()` → *Session*

Return a copy of the session.

## Testing

---

<code>Session.element_equals(other[, rtol, atol, ...])</code>	Test if each element (group, axis and array) of the current session equals the corresponding element of another session.
<code>Session.equals(other[, rtol, atol, nans_equal])</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(other, rtol=0, atol=0, nans_equal=False)` → *Array*

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.

##### **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 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.

#### Returns

##### **Boolean Array**

See also:

`Session.equals`

## Notes

Metadata is ignored.

For finite values, `element_equals` uses the following equation to test whether two arrays are equal:

$$\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$$

## 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
```

Test equality between two arrays within a given tolerance range. Return True if `absolute(s1.arr1 - s2.arr1) <= (atol + rtol * absolute(s2.arr1))`.

```
>>> s1.arr1 = Array([6., 8.], "a=a0,a1")
>>> s2.arr1 = Array([5.999, 8.001], "a=a0,a1")
```

```
>>> s1.element_equals(s2)
name      a    a01  arr1  arr2
      True  True  False  True
>>> s1.element_equals(s2, atol=0.01)
name      a    a01  arr1  arr2
      True  True  True   True
>>> s1.element_equals(s2, rtol=0.01)
name      a    a01  arr1  arr2
      True  True  True   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(other, rtol=0, atol=0, nans_equal=False) → bool`

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.

#### **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 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.

### Returns

**True** if elements of both sessions are all equal, **False** otherwise.

See also:

[`Session.element\_equals`](#)

## Notes

Metadata is ignored.

For finite values, equals uses the following equation to test whether two arrays are equal:

$$\text{absolute}(\text{array1} - \text{array2}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{array2}))$$

## 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
```

Test equality between two arrays within a given tolerance range. Return True if  $\text{absolute}(s1.\text{arr1} - s2.\text{arr1}) \leq (\text{atol} + \text{rtol} * \text{absolute}(s2.\text{arr1}))$ .

```
>>> s1.arr1 = Array([6., 8.], "a=a0,a1")
>>> s2.arr1 = Array([5.999, 8.001], "a=a0,a1")
```

```
>>> s1.equals(s2)
False
>>> s1.equals(s2, atol=0.01)
True
>>> s1.equals(s2, rtol=0.01)
True
```

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(key[, default])`

Return the object corresponding to the key.

---

## larray.Session.get

`Session.get(key, default=None) → Any`

Return 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(*args, **kwargs)</code>	Deprecated.
<code>Session.update([other])</code>	Update the session with the key/value pairs from other or passed keyword arguments, overwriting existing keys.
<code>Session.apply(func, *args[, kind])</code>	Apply function <i>func</i> on elements of the session and return a new session.
<code>Session.transpose(*args)</code>	Reorder axes of arrays in session, ignoring missing axes for each array.

**larray.Session.add**

`Session.add(*args, **kwargs) → None`

Deprecated. Please use `Session.update` instead.

**larray.Session.update**

`Session.update(other=None, **kwargs) → None`

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
['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']
```

(continues on next page)

(continued from previous page)

```
>>> 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(func, *args, kind=<class 'larray.core.array.Array'>, **kwargs) → Session`

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(*args) → Session`

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</code> ([pattern, kind])	Return a new session with objects which match some criteria.
<code>Session.compact</code> ([display])	Detect and remove "useless" axes (ie axes for which values are constant over the whole axis) for all array objects in session.

---

### larray.Session.filter

`Session.filter`(*pattern=None, kind=None*) → *Session*

Return 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(display=False) → Session`

Detect and remove “useless” axes (ie axes for which values are constant over the whole axis) for all array objects in session.

### Parameters

#### display

[bool, optional] Whether 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(fname[, names, engine, display])</code>	Load objects from a file, or several .csv files.
<code>Session.save(fname[, names, engine, ...])</code>	Dump objects from the current session to a file, or several .csv files.
<code>Session.to_csv(fname[, names, display])</code>	Dump Array objects from the current session to CSV files.
<code>Session.to_excel(fname[, names, overwrite, ...])</code>	Dump Array objects from the current session to an Excel file.
<code>Session.to_hdf(fname[, names, overwrite, ...])</code>	Dump objects from the current session to an HDF file.
<code>Session.to_pickle(fname[, names, overwrite, ...])</code>	Dump objects from the current session to a file using pickle.

## larray.Session.load

`Session.load(fname, names=None, engine='auto', display=False, **kwargs) → None`

Load objects from a file, or several .csv files. The Excel and CSV formats can only contain objects of Array type (plus metadata).

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 or Path] 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 to display which file is being worked on. Defaults to False.

## Examples

In one module:

```
>>> # scalars
>>> i, s = 5, 'string'
>>> # 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)
>>> ses = Session({'i': i, 's': s, 'a': a, 'b': b, 'a01': a01,
...               'arr1': arr1, 'arr2': arr2})
>>> # metadata
>>> ses.meta.title = 'my title'
>>> ses.meta.author = 'John Smith'
>>> # save the session in an HDF5 file
>>> ses.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
>>> ses = Session('input.h5')
>>> ses
Session(a, a01, arr1, arr2, b, i, s)
>>> ses.meta
```

(continues on next page)



(continued from previous page)

```
title: my title
author: John Smith
```

Load only some objects

```
>>> ses = Session()
>>> ses.load('input.h5', names=['s', 'a', 'b', 'arr1', 'arr2'], display=True)
opening input.h5
loading Axis object a ... done
loading Array object arr1 ... done
loading Array object arr2 ... done
loading Axis object b ... done
loading str object s ... done
```

Using .csv files (assuming the same session as above)

```
>>> ses.save('data')
>>> ses = Session()
>>> # load all .csv files from the 'data' directory
>>> ses.load('data', display=True)
opening data
loading Array object arr1 ... done
loading Array object arr2 ... done
>>> # or only arrays containing the character '1' in their names
>>> ses.load('data/*1.csv', display=True)
opening data/*1.csv
loading Array object arr1 ... done
```

## larray.Session.save

`Session.save(fname, names=None, engine='auto', overwrite=True, display=False, **kwargs) → None`

Dump objects from the current session to a file, or several .csv files. The Excel and CSV formats only dump objects of Array type (plus metadata).

### Parameters

#### fname

[str or Path] 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 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 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 to display which file is being worked on. Defaults to False.

## Examples

```
>>> # scalars
>>> i, s = 5, 'string'
>>> # 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)
>>> ses = Session({'i': i, 's': s, 'a': a, 'b': b, 'a01': a01,
...               'arr1': arr1, 'arr2': arr2})
>>> # metadata
>>> ses.meta.title = 'my title'
>>> ses.meta.author = 'John Smith'
```

Save all objects

```
>>> ses.save('output.h5', display=True)
dumping i ... done
dumping s ... done
dumping a ... done
dumping b ... done
dumping a01 ... done
dumping arr1 ... done
dumping arr2 ... done
```

Save only some objects

```
>>> ses.save('output.h5', names=['s', 'a', 'b', 'arr1', 'arr2'], display=True)
dumping s ... done
dumping a ... done
dumping b ... done
dumping arr1 ... done
dumping arr2 ... done
```

Update file

```
>>> arr1, arr4 = ndtest((3, 3)), ndtest((2, 3))
>>> ses2 = Session({'arr1': arr1, 'arr4': arr4})
>>> # replace arr1 and add arr4 in file output.h5
>>> ses2.save('output.h5', overwrite=False, display=True)
dumping arr1 ... done
dumping arr4 ... done
```

## `larray.Session.to_csv`

`Session.to_csv(fname, names=None, display=False, **kwargs) → None`

Dump 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 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

```
>>> # scalars
>>> i, s = 5, 'string'
>>> # 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)
>>> ses = Session({'i': i, 's': s, 'a': a, 'b': b, 'a01': a01,
...               'arr1': arr1, 'arr2': arr2})
>>> # metadata
>>> ses.meta.title = 'my title'
>>> ses.meta.author = 'John Smith'
```

Save all arrays (and arrays only)

```
>>> ses.to_csv('output', display=True)
dumping i ... Cannot dump i. int is not a supported type
dumping s ... Cannot dump s. str is not a supported type
dumping a ... Cannot dump a. Axis is not a supported type
dumping b ... Cannot dump b. Axis is not a supported type
dumping a01 ... Cannot dump a01. LGroup is not a supported type
dumping arr1 ... done
dumping arr2 ... done
```

Save only some arrays

```
>>> ses.to_csv('output', names=['arr1'], display=True)
dumping arr1 ... done
```

## `larray.Session.to_excel`

`Session.to_excel(fname, names=None, overwrite=True, display=False, **kwargs) → None`

Dump 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 to overwrite an existing file, if any. If False, file is updated. Defaults to True.

#### **display**

[bool, optional] Whether 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

```
>>> # scalars
>>> i, s = 5, 'string'
>>> # 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)
>>> ses = Session({'i': i, 's': s, 'a': a, 'b': b, 'a01': a01,
...               'arr1': arr1, 'arr2': arr2})
>>> # metadata
>>> ses.meta.title = 'my title'
>>> ses.meta.author = 'John Smith'
```

Save all arrays (and arrays only)

```
>>> ses.to_excel('output.xlsx', display=True)
dumping i ... Cannot dump i. int is not a supported type
dumping s ... Cannot dump s. str is not a supported type
dumping a ... Cannot dump a. Axis is not a supported type
dumping b ... Cannot dump b. Axis is not a supported type
dumping a01 ... Cannot dump a01. LGroup is not a supported type
dumping arr1 ... done
dumping arr2 ... done
```

Save only some arrays

```
>>> ses.to_excel('output.xlsx', names=['arr1'], display=True)
dumping arr1 ... done
```

## larray.Session.to\_hdf

`Session.to_hdf(fname, names=None, overwrite=True, display=False, **kwargs) → None`

Dump 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 objects to dump. Defaults to all objects present in the Session.

#### **overwrite: bool, optional**

Whether to overwrite an existing file, if any. If False, file is updated. Defaults to True.

#### **display**

[bool, optional] Whether to display which file is being worked on. Defaults to False.

## Examples

```
>>> # scalars
>>> i, s = 5, 'string'
>>> # 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)
>>> ses = Session({'i': i, 's': s, 'a': a, 'b': b, 'a01': a01,
...               'arr1': arr1, 'arr2': arr2})
>>> # metadata
>>> ses.meta.title = 'my title'
>>> ses.meta.author = 'John Smith'
```

Save all objects

```
>>> ses.to_hdf('output.h5', display=True)
dumping i ... done
dumping s ... done
dumping a ... done
dumping b ... done
dumping a01 ... done
dumping arr1 ... done
dumping arr2 ... done
```

Save only some objects

```
>>> ses.to_hdf('output.h5', names=['s', 'a', 'b', 'arr1', 'arr2'], display=True)
dumping s ... done
dumping a ... done
dumping b ... done
dumping arr1 ... done
dumping arr2 ... done
```

## larray.Session.to\_pickle

`Session.to_pickle(fname, names=None, overwrite=True, display=False, **kwargs) → None`

Dump 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 objects to dump. Defaults to all objects present in the Session.

#### **overwrite: bool, optional**

Whether to overwrite an existing file, if any. If False, file is updated. Defaults to True.

#### **display**

[bool, optional] Whether to display which file is being worked on. Defaults to False.

## Examples

```
>>> # scalars
>>> i, s = 5, 'string'
>>> # 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)
>>> ses = Session({'i': i, 's': s, 'a': a, 'b': b, 'a01': a01,
...               'arr1': arr1, 'arr2': arr2})
>>> # metadata
>>> ses.meta.title = 'my title'
>>> ses.meta.author = 'John Smith'
```

Save all objects

```
>>> ses.to_pickle('output.pkl', display=True)
dumping i ... done
dumping s ... done
dumping a ... done
dumping b ... done
```

(continues on next page)

(continued from previous page)

```
dumping a01 ... done
dumping arr1 ... done
dumping arr2 ... done
```

Save only some objects

```
>>> ses.to_pickle('output.pkl', names=['s', 'a', 'b', 'arr1', 'arr2'], display=True)
dumping s ... done
dumping a ... done
dumping b ... done
dumping arr1 ... done
dumping arr2 ... done
```

#### 4.3.14 CheckedArray

---

*CheckedArray*(axes[, dtype])

---

##### **larray.CheckedArray**

**larray.CheckedArray**(axes: ~larray.core.axis.AxisCollection, dtype: ~numpy.dtype = <class 'float'>) → Type[Array]

#### 4.3.15 CheckedSession

---

*CheckedSession*(\*args[, meta])

---

Class intended to be inherited by user defined classes in which the variables of a model are declared.

---

##### **larray.CheckedSession**

**class** **larray.CheckedSession**(\*args, meta=None, \*\*kwargs)

Class intended to be inherited by user defined classes in which the variables of a model are declared. Each declared variable is constrained by a type defined explicitly or deduced from the given default value (see examples below). All classes inheriting from *CheckedSession* will have access to all methods of the *Session* class.

The special *CheckedArray* type represents an Array object with fixed axes and/or dtype. This prevents users from modifying the dimensions (and labels) and/or the dtype of an array by mistake and make sure that the definition of an array remains always valid in the model.

By declaring variables, users will speed up the development of their models using the auto-completion (the feature in which development tools like PyCharm try to predict the variable or function a user intends to enter after only a few characters have been typed).

As for normal Session objects, it is still possible to add undeclared variables to instances of classes inheriting from *CheckedSession* but this must be done with caution.

##### **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 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.

**Warning:** The `CheckedSession.filter()`, `CheckedSession.compact()` and `CheckedSession.apply()` methods return a simple `Session` object. The type of the declared variables (and the value for the declared constants) will no longer be checked.

See also:

[\*Session\*](#), [\*CheckedParameters\*](#)

## Examples

Content of file 'parameters.py'

```
>>> from larray import *
>>> FIRST_YEAR = 2020
>>> LAST_YEAR = 2030
>>> AGE = Axis('age=0..10')
>>> GENDER = Axis('gender=male,female')
>>> TIME = Axis(f'time={FIRST_YEAR}..{LAST_YEAR}')
```

Content of file 'model.py'

```
>>> class ModelVariables(CheckedSession):
...     # --- declare variables with defined types ---
...     # Their values will be defined at runtime but must match the specified type.
...     birth_rate: Array
...     births: Array
...     # --- declare variables with a default value ---
...     # The default value will be used to set the variable if no value is passed_
...     ↪at instantiation (see below).
...     # Their type is deduced from their default value and cannot be changed at_
...     ↪runtime.
...     target_age = AGE[:2] >> '0-2'
...     population = zeros((AGE, GENDER, TIME), dtype=int)
...     # --- declare checked arrays ---
...     # The checked arrays have axes assumed to be "frozen", meaning they are
...     # constant all along the execution of the program.
...     mortality_rate: CheckedArray((AGE, GENDER))
...     # For checked arrays, the default value can be given as a scalar.
```

(continues on next page)



(continued from previous page)

```
...     # Optionally, a dtype can be also specified (defaults to float).
...     deaths: CheckedArray((AGE, GENDER, TIME), dtype=int) = 0
```

```
>>> variant_name = "baseline"
>>> # Instantiation --> create an instance of the ModelVariables class.
>>> # Warning: All variables declared without a default value must be set.
>>> m = ModelVariables(birth_rate = zeros((AGE, GENDER)),
...                   births = zeros((AGE, GENDER, TIME), dtype=int),
...                   mortality_rate = 0)
```

```
>>> # ==== model ====
>>> # In the definition of ModelVariables, the 'birth_rate' variable, has been
    ↳ declared as an Array object.
>>> # This means that the 'birth_rate' variable will always remain of type Array.
>>> # Any attempt to assign a non-Array value to 'birth_rate' will make the program
    ↳ to crash.
>>> m.birth_rate = Array([0.045, 0.055], GENDER)      # OK
>>> m.birth_rate = [0.045, 0.055]                     # Fails
Traceback (most recent call last):
...
pydantic.errors.ArbitraryTypeError: instance of Array expected
>>> # However, the arrays 'birth_rate', 'births' and 'population' have not been
    ↳ declared as 'CheckedArray'.
>>> # Thus, axes and dtype of these arrays are not protected, leading to
    ↳ potentially unexpected behavior
>>> # of the model.
>>> # example 1: Let's say we want to calculate the new births for the year 2025 and
    ↳ we assume that
>>> # the birth rate only differ by gender.
>>> # In the line below, we add an additional TIME axis to 'birth_rate' while it was
    ↳ initialized
>>> # with the AGE and GENDER axes only
>>> m.birth_rate = full((AGE, GENDER, TIME), fill_value=Array([0.045, 0.055],
    ↳ GENDER))
>>> # here 'new_births' have the AGE, GENDER and TIME axes instead of the AGE and
    ↳ GENDER axes only
>>> new_births = m.population['female', 2025] * m.birth_rate
>>> print(new_births.info)
11 x 2 x 11
age [11]: 0 1 2 ... 8 9 10
gender [2]: 'male' 'female'
time [11]: 2020 2021 2022 ... 2028 2029 2030
dtype: float64
memory used: 1.89 Kb
>>> # and the line below will crash
>>> m.births[2025] = new_births
Traceback (most recent call last):
...
ValueError: Value {time} axis is not present in target subset {age, gender}.
A value can only have the same axes or fewer axes than the subset being targeted
>>> # now let's try to do the same for deaths and making the same mistake as for
```

(continues on next page)

(continued from previous page)

```

→'birth_rate'.
>>> # The program will crash now at the first step instead of letting you go further
>>> m.mortality_rate = full((AGE, GENDER, TIME), fill_value=sequence(AGE, inc=0.02))
Traceback (most recent call last):
...
ValueError: Array 'mortality_rate' was declared with axes {age, gender} but got
→array with axes
{age, gender, time} (unexpected {time} axis)

```

```

>>> # example 2: let's say we want to calculate the new births for all years.
>>> m.birth_rate = full((AGE, GENDER, TIME), fill_value=Array([0.045, 0.055],
→GENDER))
>>> new_births = m.population['female'] * m.birth_rate
>>> # here 'new_births' has the same axes as 'births' but is a float array instead of
>>> # an integer array as 'births'.
>>> # The line below will make the 'births' array become a float array while
>>> # it was initialized as an integer array
>>> m.births = new_births
>>> print(m.births.info)
11 x 11 x 2
age [11]: 0 1 2 ... 8 9 10
time [11]: 2020 2021 2022 ... 2028 2029 2030
gender [2]: 'male' 'female'
dtype: float64
memory used: 1.89 Kb
>>> # now let's try to do the same for deaths.
>>> m.mortality_rate = full((AGE, GENDER), fill_value=sequence(AGE, inc=0.02))
>>> # here the result of the multiplication of the 'population' array by the
→'mortality_rate' array
>>> # is automatically converted to an integer array
>>> m.deaths = m.population * m.mortality_rate
>>> print(m.deaths.info)
11 x 2 x 11
age [11]: 0 1 2 ... 8 9 10
gender [2]: 'male' 'female'
time [11]: 2020 2021 2022 ... 2028 2029 2030
dtype: int32
memory used: 968 bytes

```

It is possible to add undeclared variables to a checked session but this will print a warning:

```

>>> m.undeclared_var = 'my_value'
UserWarning: 'undeclared_var' is not declared in 'ModelVariables'

```

```

>>> # ==== output ====
>>> # save all variables in an HDF5 file
>>> m.save(f'{variant_name}.h5', display=True)
dumping birth_rate ... done
dumping births ... done
dumping mortality_rate ... done
dumping deaths ... done
dumping target_age ... done

```

(continues on next page)

(continued from previous page)

```
dumping population ... done
dumping undeclared_var ... done
```

```
__init__(*args, meta=None, **kwargs)
```

## Methods

<code>__init__(*args[, meta])</code>	
<code>add(*args, **kwargs)</code>	Deprecated.
<code>apply(func, *args[, kind])</code>	Apply function <i>func</i> on elements of the session and return a new session.
<code>array_equals(**kwargs)</code>	
<code>compact([display])</code>	Detect and remove "useless" axes (ie axes for which values are constant over the whole axis) for all array objects in session.
<code>copy()</code>	Return a copy of the session.
<code>dict([exclude])</code>	
<code>dump(**kwargs)</code>	
<code>dump_csv(**kwargs)</code>	
<code>dump_excel(**kwargs)</code>	
<code>dump_hdf(**kwargs)</code>	
<code>element_equals(other[, rtol, atol, nans_equal])</code>	Test if each element (group, axis and array) of the current session equals the corresponding element of another session.
<code>equals(other[, rtol, atol, nans_equal])</code>	Test if all elements (groups, axes and arrays) of the current session are equal to those of another session.
<code>filter([pattern, kind])</code>	Return a new session with objects which match some criteria.
<code>get(key[, default])</code>	Return the object corresponding to the key.
<code>items()</code>	Return a view of the session's items ((key, value) pairs).
<code>keys()</code>	Return a view on the session's keys.
<code>load(fname[, names, engine, display])</code>	Load objects from a file, or several .csv files.
<code>save(fname[, names, engine, overwrite, display])</code>	Dump objects from the current session to a file, or several .csv files.
<code>summary([template])</code>	Return a summary of the content of the session.
<code>to_csv(fname[, names, display])</code>	Dump Array objects from the current session to CSV files.
<code>to_excel(fname[, names, overwrite, display])</code>	Dump Array objects from the current session to an Excel file.
<code>to_globals([names, depth, warn, inplace])</code>	Create global variables out of objects in the session.
<code>to_hdf(fname[, names, overwrite, display])</code>	Dump objects from the current session to an HDF file.
<code>to_pickle(fname[, names, overwrite, display])</code>	Dump objects from the current session to a file using pickle.
<code>transpose(*args)</code>	Reorder axes of arrays in session, ignoring missing axes for each array.
<code>update([other])</code>	Update the session with the key/value pairs from other or passed keyword arguments, overwriting existing keys.
<code>values()</code>	Return a view on the session's values.

## Attributes

<code>memory_used</code>	Return the memory consumed by the session in human readable form.
<code>meta</code>	Return metadata of the session.
<code>names</code>	Return the list of names of the objects in the session.
<code>nbytes</code>	Return the memory in bytes consumed by the session.

### 4.3.16 CheckedParameters

<code>CheckedParameters(*args[, meta])</code>	Same as <code>py:class:CheckedSession</code> but declared variables cannot be modified after initialization.
---	--

## `larray.CheckedParameters`

**class** `larray.CheckedParameters(*args, meta=None, **kwargs)`

Same as `py:class:CheckedSession` but declared variables cannot be modified after initialization.

### 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 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.

See also:

[`CheckedSession`](#)

## Examples

Content of file 'parameters.py'

```
>>> from larray import *
>>> class Parameters(CheckedParameters):
...     # --- declare variables with fixed values ---
...     # The given values can never be changed
...     FIRST_YEAR = 2020
...     LAST_YEAR = 2030
...     AGE = Axis('age=0..10')
...     GENDER = Axis('gender=male,female')
...     TIME = Axis(f'time={FIRST_YEAR}..{LAST_YEAR}')
...     # --- declare variables with defined types ---
```

(continues on next page)

(continued from previous page)

```
...     # Their values must be defined at initialized and will be frozen after.
...     variant_name: str
```

Content of file ‘model.py’

```
>>> # instantiation --> create an instance of the ModelVariables class
>>> # all variables declared without value must be set
>>> P = Parameters(variant_name='variant_1')
>>> # once an instance is created, its variables can be accessed but not modified
>>> P.variant_name
'variant_1'
>>> P.variant_name = 'new_variant'
Traceback (most recent call last):
...
TypeError: Cannot change the value of the variable 'variant_name' since 'Parameters'
is immutable and does not support item assignment
```

```
__init__(*args, meta=None, **kwargs)
```

## Methods

<code>__init__(*args[, meta])</code>	
<code>add(*args, **kwargs)</code>	Deprecated.
<code>apply(func, *args[, kind])</code>	Apply function <i>func</i> on elements of the session and return a new session.
<code>array_equals(**kwargs)</code>	
<code>compact([display])</code>	Detect and remove "useless" axes (ie axes for which values are constant over the whole axis) for all array objects in session.
<code>copy()</code>	Return a copy of the session.
<code>dict([exclude])</code>	
<code>dump(**kwargs)</code>	
<code>dump_csv(**kwargs)</code>	
<code>dump_excel(**kwargs)</code>	
<code>dump_hdf(**kwargs)</code>	
<code>element_equals(other[, rtol, atol, nans_equal])</code>	Test if each element (group, axis and array) of the current session equals the corresponding element of another session.
<code>equals(other[, rtol, atol, nans_equal])</code>	Test if all elements (groups, axes and arrays) of the current session are equal to those of another session.
<code>filter([pattern, kind])</code>	Return a new session with objects which match some criteria.
<code>get(key[, default])</code>	Return the object corresponding to the key.
<code>items()</code>	Return a view of the session's items ((key, value) pairs).
<code>keys()</code>	Return a view on the session's keys.
<code>load(fname[, names, engine, display])</code>	Load objects from a file, or several .csv files.
<code>save(fname[, names, engine, overwrite, display])</code>	Dump objects from the current session to a file, or several .csv files.
<code>summary([template])</code>	Return a summary of the content of the session.
<code>to_csv(fname[, names, display])</code>	Dump Array objects from the current session to CSV files.
<code>to_excel(fname[, names, overwrite, display])</code>	Dump Array objects from the current session to an Excel file.
<code>to_globals([names, depth, warn, inplace])</code>	Create global variables out of objects in the session.
<code>to_hdf(fname[, names, overwrite, display])</code>	Dump objects from the current session to an HDF file.
<code>to_pickle(fname[, names, overwrite, display])</code>	Dump objects from the current session to a file using pickle.
<code>transpose(*args)</code>	Reorder axes of arrays in session, ignoring missing axes for each array.
<code>update([other])</code>	Update the session with the key/value pairs from other or passed keyword arguments, overwriting existing keys.
<code>values()</code>	Return a view on the session's values.

## Attributes

<code>memory_used</code>	Return the memory consumed by the session in human readable form.
<code>meta</code>	Return metadata of the session.
<code>names</code>	Return the list of names of the objects in the session.
<code>nbytes</code>	Return the memory in bytes consumed by the session.

## 4.3.17 Editor

<code>view([obj, title, depth])</code>	Open a new viewer window.
<code>edit([obj, title, minvalue, maxvalue, ...])</code>	Open a new editor window.
<code>debug([depth])</code>	Open a new debug window.
<code>compare(*args[, depth])</code>	Open a new comparator window, comparing arrays or sessions.
<code>run_editor_on_exception([root_path, ...])</code>	Run the editor when an unhandled exception (a fatal error) happens.

## `larray.view`

`larray.view(obj=None, title="", depth=0)`

Open a new viewer window. Arrays are loaded in readonly mode and their content cannot be modified.

### Parameters

#### **obj**

[`np.ndarray`, `Array`, `Session`, `dict`, `str` or `Path`, optional] Object to visualize. If string or `Path`, 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)`

Open a new editor window.

### Parameters

#### **obj**

[np.ndarray, Array, Session, dict, str, Path, REOPEN\_LAST\_FILE or None, optional] Object to visualize. If string or Path, array(s) will be loaded from the file given as argument. Passing the constant REOPEN\_LAST\_FILE loads the last opened file. Defaults to None, which gathers all variables (global and local) 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 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)`

Open 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, depth=0, **kwargs)`

Open a new comparator window, comparing arrays or sessions.

### Parameters

#### **\*args**

[Arrays, Sessions, str or Path.] Arrays or sessions to compare. Strings or Path will be loaded as Sessions from the corresponding files.

#### **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.

#### **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 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.

#### **depth**

[int, optional] Stack depth where to look for variables. Defaults to 0 (where this function was called).

## 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)`

Run 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 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 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`

## 4.3.18 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>	Generate a random sample from given choices.

## `larray.random.randint`

`larray.random.randint(low, high=None, axes=None, dtype='l', meta=None) → Array`

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 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
```

With varying low and high (each depending on a different axis)

```
>>> low = la.sequence('a=a0,a1')
>>> low
a  a0  a1
   0  1
>>> high = la.sequence('b=b0..b2', initial=3)
>>> high
b  b0  b1  b2
   3  4  5
```

In other words, we want to generate values between low and high (high included) for each cell. Let's note that `low..high`:

**ab b0 b1 b2**

a0 0..2 0..3 0..4 a1 1..2 1..3 1..4

```
>>> la.random.randint(low, high)
a\b  b0  b1  b2
a0    0   2   2
a1    2   3   4
```

### larray.random.normal

`larray.random.normal(loc=0.0, scale=1.0, axes=None, meta=None) → Array`

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 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  $\mu$  is the mean and  $\sigma$  the standard deviation. The square of the standard deviation,  $\sigma^2$ , 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)

```
>>> mu = la.sequence('a=a0,a1', initial=5, inc=5)
>>> mu
a a0 a1
  5 10
>>> sigma = la.sequence('b=b0..b2', 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))
```

(continues on next page)

(continued from previous page)

```
>>> _ = 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) → Array`

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 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)

```
>>> low = la.sequence('a=a0,a1')
>>> low
a  a0  a1
   0   1
>>> high = la.sequence('b=b0..b2', 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) → Array`

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) → Array`

Generate 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 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
```

### 4.3.19 Constants

<code>nan</code>	NaN (Not a Number)
<code>inf</code>	$\infty$ (infinite)
<code>pi</code>	$\pi$
<code>e</code>	$e$
<code>euler_gamma</code>	Euler's $\gamma$

### **larray.core.constants.nan**

`larray.core.constants.nan = nan`  
NaN (Not a Number)

### **larray.core.constants.inf**

`larray.core.constants.inf = inf`  
 $\infty$  (infinite)

### **larray.core.constants.pi**

`larray.core.constants.pi = 3.141592653589793`  
 $\pi$

### **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  $\gamma$

## INDICES AND TABLES

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



## 6.1 Change log

### 6.1.1 Version 0.34.2

Released on 2023-10-23.

#### CORE

##### New features

- added support for evaluating expressions using `X.axis_name` when calling some built-in functions, most notably `where()`. For example, the following code now works (previously it seemed to work but produced the wrong result – see the fixes section below):

```
>>> arr = ndtest("age=0..3")
>>> arr
age  0  1  2  3
     0  1  2  3
>>> where(X.age == 2, 42, arr)
age  0  1  2  3
     0  1 42  3
```

##### Fixes

- fixed `Array.reindex` when using an axis object from the array as `axes_to_reindex` (closes [issue 1088](#)).
- fixed `Array.reindex({axis: list_of_labels})` (closes [issue 1068](#)).
- `Array.split_axes` now raises an explicit error when some labels contain more separators than others, instead of silently dropping part of those labels, or even some data (closes [issue 1089](#)).
- a boolean condition including only `X.axis_name` and scalars (e.g. `X.age == 0`) raises an error when Python needs to know whether it is True or not (because there is no array to extract the axis labels from) instead of always evaluating to True. This was especially dangerous in the context of a `where()` function, which always evaluated to its left side (e.g. `where(X.age > 0, arr, 0)` evaluated to `arr` for all ages). Closes [issue 1083](#).
- expressions using `X.axis_name` and an `Array` now evaluate correctly when the `Array` is not involved in the first operation. For example, this already worked:

```
>>> arr = ndtest("age=0..3")
>>> arr
age  0  1  2  3
     0  1  2  3
>>> arr * (X.age != 2)
age  0  1  2  3
     0  1  0  3
```

but this did not:

```
>>> (X.age != 2) * arr
```

- fixed plots with fewer than 6 integer labels in the x axis. In that case, it interpolated the values, which usually looks wrong for integer labels (e.g. year). Closes [issue 1076](#).

## EDITOR

### Fixes

- fixed the viewer being unusable after showing a matplotlib plot (closes [issue 261](#)).
- silence spurious debugger warning on Python 3.11 (closes [issue 263](#)).
- when code in the interactive console creates *and shows* a plot window, avoid showing it a second time (closes [issue 265](#)).
- depending on the system regional settings, comparator tolerance sometimes did not allow simple fractional numbers (e.g. 0.1). The only way to specify the tolerance was the scientific notation (closes [issue 260](#)).

## 6.1.2 Version 0.34.1

Released on 2023-09-14.

## CORE

### New features

- added support for Python 3.11.
- added support for stacking all arrays of a Session by simply doing: `stack(my_session)` instead of `stack(my_session.items())` (closes [issue 1057](#)).

### Fixes

- avoid warnings with recent versions of Pandas or Numpy (closes [issue 1061](#)).



## EDITOR

### New features

- added support for Python 3.11.

### Fixes

- fixed viewer being unusable in a script run via PyCharm “run with console” (closes [issue 253](#)).
- fixed keyboard navigation when using PyQt6 (closes [issue 235](#)).

## 6.1.3 Version 0.34

Released on 2023-03-14.

## CORE

### Syntax changes

- made `Array.append()` work for the cases previously covered by `Array.extend()` (when the appended value already has the axis being extended) and deprecated `Array.extend()` (closes [issue 887](#)).
- renamed `Array.sort_axes()` to `Array.sort_labels()` (closes [issue 861](#)).
- renamed `Array.percentile()` and `Array.percentile_by()` *interpolation* argument to *method* to follow numpy and thus support additional “interpolation” methods.
- deprecated the ability to target a label in an aggregated array using the group that created it. The aggregated array label should be used instead. This is a seldom used feature which is complex to keep working and has a significant performance cost in some cases, even when the feature is not used (closes [issue 994](#)).

In other words, the following code will now raise a warning:

```
>>> arr = ndtest(4)
>>> arr
a  a0  a1  a2  a3
   0   1   2   3
>>> group1 = arr.a['a0', 'a2'] >> 'a0_a2'
>>> group2 = arr.a['a1', 'a3'] >> 'a1_a3'
>>> agg_arr = arr.sum((group1, group2))
>>> agg_arr
a  a0_a2  a1_a3
   2       4
>>> agg_arr[group1]
FutureWarning: Using a Group object which was used to create an aggregate to target_
↳ its aggregated label is deprecated.
Please use the aggregated label directly instead. In this case, you should use 'a0_
↳ a2' instead of using
a['a0', 'a2'] >> 'a0_a2'.
2
```

One should use the label on the aggregated array instead:

```
>>> agg_arr['a0_a2']  
2
```

- deprecated passing individual session elements as non-keyword arguments to `Session()`. This means that, for example, `Session(axis1, axis2, array1=array1)` should be rewritten as `Session(axis1name=axis1, axis2name=axis2, array1=array1)` instead. Closes [issue 1024](#).
- deprecated `Session.add()`. Please use `Session.update()` instead (closes [issue 999](#)).

## Backward incompatible changes

- dropped support for Python 3.6.
- deprecations dating to version 0.29 or earlier (released more than 3 years ago) now raise errors instead of printing a warning.

## New features

- added support for Python 3.10.
- implemented `Array.value_counts()`, which computes the number of occurrences of each unique value in an array.
- added `Session.nbytes` and added `Session.memory_used` attributes.
- added `display` argument to `Array.compact()` to display a message if some axes were “compacted”.

## Miscellaneous improvements

- made all I/O functions/methods/constructors accept `pathlib.Path` objects in addition to strings for all arguments representing a path (closes [issue 896](#)).
- added type hints for all remaining functions and methods which improves autocompletion in editors (such as PyCharm). Closes [issue 864](#).
- made several error messages more useful when trying to get an invalid subset of an array (closes [issue 875](#)).
  - when a key is not valid on any axis, the error message includes the array axes
  - when a key is not valid for the axis specified by the user, the error message includes that axis labels
  - when a label is ambiguous (valid on several axes), the error message includes the axes labels in addition to the axes names
  - when several parts of a key seem to target the same axis, the error message includes the bad key in addition to the axis.
- made `ipfp()` faster (the smaller the array, the larger the improvement) For example, for small arrays it is several times faster than before, for 1000x1000 arrays it is about 30% faster.
- made arithmetic operations between two Arrays with the same axes much faster.
- made `Array[]` faster in the usual/simple cases.
- made `Array.i[]` much faster.

## Fixes

- fixed displaying plots made via `Array.plot()` outside of the LArray editor (closes [issue 1019](#)).
- fixed `Array.insert()` when no label is provided (closes [issue 879](#)).
- fixed `Array.insert()` when (one of) the inserted label(s) is ambiguous on the value.
- fixed comparison between `Array` and `None` returning False instead of an array of boolean values (closes [issue 988](#))
- fixed binary operations between an `Array` and an `Axis` returning False.
- fixed `AxisCollection.split_axes()` with anonymous axes.
- fixed the names argument in `Array.split_axes()` and `AxisCollection.split_axes()` not working in some cases.
- fixed taking a subset of an Excel range (e.g. `myworkbook['my_sheet']['A2:C5'][1:]`)
- fixed setting the first sheet via position in a new workbook opened via `open_excel(overwrite=True)`:

```
>>> with open_excel(fpath, overwrite_file=True) as wb:
...     wb[0] = <something>
```

- fixed `Array.ipoints[]` when not all dimensions are given in the key.

## EDITOR

### New features

- added support for Python 3.10.
- the initial column width is now set depending on the content and resized automatically when changing the number of digits (closes [issue 145](#)).

### Miscellaneous improvements

- plot windows title now include the expression used to make the plot (the name of the array in most cases) (closes [issue 233](#)).
- when displaying an expression (computed array), the window title includes the actual expression instead of using `<expr>`.
- `compare()` can now take filepaths as argument (and will load them as a Session) to make comparing a in-memory Session with an earlier Session saved on the disk. Those filepaths can be given as both str or Path objects. Closes [issue 229](#).
- added support for Path objects (in addition to str representing paths) in `view()` and `edit()`. See [issue 896](#).
- when the editor displays currently defined variables (via `debug()` `edit()` or `view()` without argument within user code or via an exception when `run_editor_on_exception` is active), LArray functions are not made available systematically in the console anymore (what is available in the console is really what was available in the users code). This closes [issue 199](#).
- added support for incomplete slices in “save command history”, like in Python slices. For example, one can save from line 10 onwards by using “10:” or “10..”, i.e. without specifying the last line. See [issue 225](#).

### Fixes

- fixed `run_editor_on_exception()` so that the larray editor is not opened when trying to stop a program (via Ctrl-C or the IDE stop button). Closes [issue 231](#).
- improved the situation when trying to stop a program (via *Ctrl-C* or the IDE stop button) with an LArray Editor window open. It used to ignore such events altogether, forcing the IDE to send a “kill” event when pressing the button a second time, which could leave some resource open (e.g Excel instances). Now, the LArray Editor will close itself when its parent program is asked to stop *but* so far, it will only do so when the window is active again. This makes for an odd behavior but at least cleans up the program properly (closes [issue 231](#)).
- when save command history fails, do not do so silently. Closes [issue 225](#).
- fixed saving command history to a path containing spaces. Closes [issue 244](#).
- fixed `compare()` background color being red for two equal integer arrays instead of white (closes [issue 246](#)).

### 6.1.4 Version 0.33.1

Released on 2021-09-22.

### CORE

#### Miscellaneous improvements

- added type hints for many Array methods (see [issue 864](#)) which improves autocompletion in editors (such as PyCharm).

### Fixes

- fixed `CheckedSession` with pydantic version >1.5 (closes [issue 958](#)).
- removed the constraint on pydantic version in `larrayenv`, making it actually installable.
- fixed using labels for x and y in `Array.plot()` and `Array.plot.scatter()` functions, as well as `Array.plot.pie()` (closes [issue 969](#)).
- fixed wrong “source code line” in “field is not declared” warning in `CheckedSession.__init__()` (closes [issue 968](#)).
- fixed `Array.growth_rate()` returning nans instead of zeros when consecutive values are zeros (closes [issue 903](#)).

### EDITOR

#### Fixes

- fixed autocompletion in the editor console when using some particular dependencies versions (closes [issue 220](#)).
- fixed the “Update LArray” shortcut in Windows start menu when LArray is not installed in the “base” conda environment (closes [issue 211](#)).

## 6.1.5 Version 0.33

Released on 2021-08-17.

### CORE

#### New features

- added official support for Python 3.9 (0.32.3 already supports it even though it was not mentioned).
- added *CheckedSession*, *CheckedParameters* and *CheckedArray* objects.

*CheckedSession* is intended to be inherited by user defined classes in which the variables of a model are declared. By declaring variables, users will speed up the development of their models using the auto-completion (the feature in which development tools like PyCharm try to predict the variable or function a user intends to enter after only a few characters have been typed). All user defined classes inheriting from *CheckedSession* will have access to the same methods as *Session* objects.

*CheckedParameters* is the same as *CheckedSession* but the declared variables cannot be modified after initialization.

The special *CheckedArray* type represents an Array object with fixed axes and/or dtype. It is intended to be only used along with *CheckedSession*.

Closes [issue 832](#).

#### Miscellaneous improvements

- greatly improved *Array.plot()* method and “submethods” (*Array.plot.bar()*, etc.)
  - support *x*, *y* and *by* arguments in plot functions where it make sense When only some of them are specified, the other arguments pick from remaining available axes. This means a lot of plots can now be expressed more intuitively and concisely (you do not need to transpose your array to get the result you want, you just specify the axes you want to use in ‘x’ or ‘y’.
  - *subplots* argument now accepts an axis (or tuple of them) in addition to a boolean to specify *which* axes to use as subplots.
  - support for *labels* (instead of axes) in x and y for line plot and scatter.
  - support passing a dict as legend to customize the legend.
  - many tweaks to make several plots look better out of the box.
- *eye()* now supports an *AxisCollection* as argument, so you can use axes from another array by using *eye(other\_array.axes)*.
- added arguments *rtol*, *atol* and *nans\_equal* to the *Session.element\_equals()* and *Session.equals()* methods (closes [issue 990](#)).

### Fixes

- fixed `Array.values()`, `zip_array_values` and `zip_array_items` when `axes=()` (closes [issue 883](#)).
- fixed several edge cases in `sequence()`.
- fixed `set_labels(labels_dict)` with several labels from the same axis (closes [issue 851](#)).
- fixed loading arrays with anonymous axes and numeric labels from Excel using Pandas 1.3+ (closes [issue 950](#)).
- fixed `read_hdf()` opening in RW mode instead of read mode (closes [issue 980](#)).

### EDITOR

#### 6.1.6 Version 0.32.3

Released on 2021-06-08.

### CORE

#### Backward incompatible changes

- dropped support for Python 2 (closes [issue 567](#)).

#### New features

- added support for Python 3.8 (closes [issue 850](#)).

#### Miscellaneous improvements

- scalar objects (i.e of type int, float, bool, string, date, time or datetime) belonging to a session are now also saved and loaded when using the HDF5 or pickle format (closes [issue 842](#)).
- implemented `Axis.astype()` method (closes [issue 880](#)).
- added `min_y`, `max_y` and `xticks_spacing` keyword arguments to the `ReportSheet.add_graph()` and `ReportSheet.add_graphs()` methods (closes [issue 901](#)).
- implemented `isscalar()` function (closes [issue 872](#)).
- implemented the `Array.allclose()` method (closes [issue 871](#)).
- implemented `Axis.min()` and `Axis.max()` methods (closes [issue 874](#)).

## Fixes

- fixed an edge case for group aggregates and labels in reverse order (closes [issue 868](#)).
- fixed \*\_by functions returning the same value as if no axis was passed when specifying all axes (closes [issue 913](#))

## EDITOR

### Backward incompatible changes

- dropped Python 2 support (closes [issue 132](#)).

## Fixes

- workaround incompatibility with Python3.8 on Windows (closes [issue 208](#)).
- workaround incompatibility between two of our dependencies versions preventing the editor to even start (closes [issue 209](#)).

## 6.1.7 Version 0.32.2

Released on 2020-04-03.

## CORE

### Fixes

- fixed using Pandas >= 1.0 (closes [issue 845](#)).
- fixed the missing space between parameters name and type in API documentation (closes [issue 849](#)).
- fixed a few issues for Python 2.7 and/or Linux.

## EDITOR

### Fixes

- fixed spurious warning in the console when an expression results in an empty sequence (array, list, tuple).
- fixed displaying arrays entirely filled with NaN.

### 6.1.8 Version 0.32.1

Released on 2019-12-19.

#### CORE

##### Miscellaneous improvements

- improved the tutorial and some examples to make them more intuitive (closes [issue 829](#)).

#### Fixes

- fixed loading arrays with more than 2 dimensions but no axes names (even when specifying `nb_axes` explicitly). This case mostly occurs when trying to load a specific range of an Excel file (closes [issue 830](#) and [issue 831](#)).

#### EDITOR

##### Fixes

- fixed the “Cancel” button of the confirmation dialog when trying to quit the editor with unsaved modifications. It was equivalent to discard, potentially leading to data loss.
- fixed (harmless) error messages appearing when trying to display any variable via the console when using matplotlib 3.1+

### 6.1.9 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.

## 6.1.10 Version 0.31

Released on 2019-08-09.

## CORE

### New features

- added the `ExcelReport` class allowing to generate multiple graphs in an Excel file at once (closes [issue 676](#)).

### Fixes

- 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](#).

### EDITOR

#### New features

- implemented `run_editor_on_exception()` function. If you call this function in your code (for example at the top of your main script), Python will open an larray editor if any unexpected error happens anywhere in your script (closes [issue 180](#)).

### Fixes

- fixed exception raised when loading a session with non array objects from a file (closes [issue 179](#)).
- fixed passing a simple array to the `view()` and `edit()` functions (closes [issue 183](#)).

## 6.1.11 Version 0.30

Released on 2019-06-27.

### CORE

#### Syntax changes

- `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](#)).

## Backward incompatible changes

- 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`.

## New features

- 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.

## Miscellaneous improvements

- 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,F0;sex=M,F')
>>> a
nat\sex  M  F
      BE  0  1
      F0  2  3
>>> a.set_labels({'M': 'Men', 'BE': 'Belgian'})
nat\sex  Men  F
Belgian   0  1
      F0   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
```

(continues on next page)

(continued from previous page)

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.

## Fixes

- 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()
```

## EDITOR

### Miscellaneous improvements

- display the filename and line number in the status bar when the editor is called from a Python script (closes [issue 173](#)).

## Fixes

- fixed editor console unusable after any exception (closes [issue 166](#)).
- fixed LArray editor launcher in Windows menu (closes [issue 169](#)).

## 6.1.12 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
>>> pi
3.141592653589793
OR
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> arr[key]
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})",
```

(continues on next page)

(continued from previous page)

```

...           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)
    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
      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.

### 6.1.13 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")
```

(continues on next page)



(continued from previous page)

```
>>> 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)
```

(continues on next page)

(continued from previous page)

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
>>> 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
```

(continues on next page)

(continued from previous page)

```

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
        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
>>> 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
```

(continues on next page)

(continued from previous page)

```
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)
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  $\geq 2$ 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
```

## 6.1.14 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')
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')
```

(continues on next page)



(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```

    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,F0,BE")
>>> a
sex\nat  EU  F0  BE
      M  10   2   4
      F   3   7   1
>>> a.sort_values(axis='sex')
sex*\nat  EU  F0  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,F0,BE")
>>> a
sex\nat  EU  F0  BE
      M  10   2   4
      F   3   7   1
>>> a.sort_values()
sex_nat  F_BE  M_F0  F_EU  M_BE  F_F0  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()).

## 6.1.15 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](#)).

### 6.1.16 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, startingwith, endingwith and matching to create a group with all labels of a group matching some criterion (closes [issue 108](#)).

```
>>> group = people.startingwith('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
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',
 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
```

(continues on next page)

(continued from previous page)

```
>>> '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
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
BE  0  1
FO  2  3
>>> a.set_labels(sex='Men,Women', nat='Belgian,Foreigner')
```

(continues on next page)

(continued from previous page)

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 " ", 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
      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.

## 6.1.17 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.

### 6.1.18 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).

### 6.1.19 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
>>> 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)])
>>> 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
```

### 6.1.20 Version 0.24.1

Released on 2017-06-14.

#### Fixes

- updated the tutorial to use version 0.24 syntax.

### 6.1.21 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['gen<tab> # will be completed to `arr.axes['gender`
```

```
>>> wb = open_excel()
>>> wb['Sh<tab> # 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.

## 6.1.22 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
```

## 6.1.23 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
>>> 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'))
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 `s['a_<tab>` will be completed to `s['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](#))

## 6.1.24 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,F0;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
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](#)).

### 6.1.25 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.

## 6.1.26 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
```

(continues on next page)

(continued from previous page)

```

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(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']

```

(continues on next page)



(continued from previous page)

```
>>> 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')
```

(continues on next page)

(continued from previous page)

```
>>> agg
a\b | tob2 | other
a0 |    3 |    5
a1 |   15 |   13
a2 |   27 |   21
>>> agg.info
3 x 2
a [3]: 'a0' 'a1' 'a2'
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, ...

## 6.1.27 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
```

(continues on next page)

(continued from previous page)

```

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)

## 6.1.28 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, ...

## 6.1.29 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

### 6.1.30 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)
# 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
```

(continues on next page)

(continued from previous page)

```

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(1000000)
>>> a.memory_used
'390.62 Kb'
```

- implemented Axis + AxisCollection

```
>>> a = ndrange('sex=M,F;type=t1,t2')
>>> Axis('nat', 'BE,FO') + a.axes
```

(continues on next page)

(continued from previous page)

```
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.

### 6.1.31 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

## 6.1.32 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

## 6.1.33 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

### Excel support

- 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

## Core

- 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

## 6.1.34 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']
```

(continues on next page)



(continued from previous page)

```
# write a without labels starting at C5
>>> 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 0x1ff1bae22e8>
>>> # 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, ...

## 6.1.35 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))
```

(continues on next page)

(continued from previous page)

```

>>> a
nat\sex | M | F
      BE | 0 | 1
      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)
```

(continues on next page)

(continued from previous page)

```

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
          1 | 4 | 5 | 6 | 7
          2 | 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/...

### 6.1.36 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 *`

### 6.1.37 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 | 1.33333333333 | 0.714285714286
>>> a.growth_rate(d=2)
```

(continues on next page)

(continued from previous page)

```
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
      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
```

(continues on next page)



(continued from previous page)

```
>>> 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
```

(continues on next page)

(continued from previous page)

```

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
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'])

```

(continues on next page)

(continued from previous page)

```

>>> 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
>>> 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

```

(continues on next page)

(continued from previous page)

```
    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

## 6.1.38 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.

### 6.1.39 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

### 6.1.40 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

### 6.1.41 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

### 6.1.42 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.

### 6.1.43 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
>>> # dimension
>>> arr.i[0, :, :, 1] = 42
```

- added optional 'readonly' argument to expand() to produce a readonly view (much faster since no copying is done)

## 6.1.44 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

## 6.1.45 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

### 6.1.46 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)
- ndecimals 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

## 6.1.47 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

## 6.1.48 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

## 6.1.49 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

## 6.1.50 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

### 6.1.51 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`

### 6.1.52 Version 0.2.6

Released on 2015-11-24.

#### Fixes

- fixed LArray.cumsum and cumprod.
- fixed all doctests just enough so that they run.

### 6.1.53 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]]`).

### 6.1.54 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]]
```

### 6.1.55 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

## 6.1.56 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 <= 0`

## 6.1.57 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

## 6.1.58 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.

## 6.1.59 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 [ruff](#).

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 dependencies

```
# Create and activate the environment
conda create -n larray_dev numpy pandas pytables pyqt qtpy matplotlib openpyxl
↪xlswriter pytest
conda activate larray_dev
# Install ruff (as of September 2023, it is not available on Anaconda)
pip install ruff
```

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. Install larray in “development mode”

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!

You can check your code respects most of those conventions and some other style guidelines by running the following command in the project directory:

```
> ruff check .
```

### 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
    -----
    number : int
        The number to test.
```

(continues on next page)

(continued from previous page)

```

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/`. 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 via Github actions 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 numpydoc 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] R. J. Hyndman and Y. Fan, “Sample quantiles in statistical packages,” *The American Statistician*, 50(4), pp. 361-365, 1996
- [1] R. J. Hyndman and Y. Fan, “Sample quantiles in statistical packages,” *The American Statistician*, 50(4), pp. 361-365, 1996
- [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. [https://personal.math.ubc.ca/~cbm/aands/page\\_379.htm](https://personal.math.ubc.ca/~cbm/aands/page_379.htm)
- [3] <https://metacpan.org/pod/distribution/Math-Cephes/lib/Math/Cephes.pod#i0:-Modified-Bessel-function-of-order-zero>
- [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](https://en.wikipedia.org/wiki/Sinc_function)
- [1] “Lecture Notes on the Status of IEEE 754”, William Kahan, <https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>
- [1] Wikipedia, “Exponential function”, [https://en.wikipedia.org/wiki/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, [https://personal.math.ubc.ca/~cbm/aands/page\\_69.htm](https://personal.math.ubc.ca/~cbm/aands/page_69.htm)
- [1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. [https://personal.math.ubc.ca/~cbm/aands/page\\_67.htm](https://personal.math.ubc.ca/~cbm/aands/page_67.htm)
- [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. [https://personal.math.ubc.ca/~cbm/aands/page\\_67.htm](https://personal.math.ubc.ca/~cbm/aands/page_67.htm)
- [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. [https://personal.math.ubc.ca/~cbm/aands/page\\_67.htm](https://personal.math.ubc.ca/~cbm/aands/page_67.htm)
- [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. [https://personal.math.ubc.ca/~cbm/aands/page\\_83.htm](https://personal.math.ubc.ca/~cbm/aands/page_83.htm)

- [2] Wikipedia, “Hyperbolic function”, [https://en.wikipedia.org/wiki/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. [https://personal.math.ubc.ca/~cbm/aands/page\\_86.htm](https://personal.math.ubc.ca/~cbm/aands/page_86.htm)
- [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. [https://personal.math.ubc.ca/~cbm/aands/page\\_86.htm](https://personal.math.ubc.ca/~cbm/aands/page_86.htm)
- [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. [https://personal.math.ubc.ca/~cbm/aands/page\\_86.htm](https://personal.math.ubc.ca/~cbm/aands/page_86.htm)
- [2] Wikipedia, “Inverse hyperbolic function”, <https://en.wikipedia.org/wiki/Arctanh>
- [1] Wikipedia, “Normal distribution”, [http://en.wikipedia.org/wiki/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\\_\\_\(\) \(larray.Axis method\), 117](#)  
[\\_\\_init\\_\\_\(\) \(larray.Array method\), 175](#)  
[\\_\\_init\\_\\_\(\) \(larray.Axis method\), 110](#)  
[\\_\\_init\\_\\_\(\) \(larray.AxisCollection method\), 150](#)  
[\\_\\_init\\_\\_\(\) \(larray.CheckedParameters method\), 474](#)  
[\\_\\_init\\_\\_\(\) \(larray.CheckedSession method\), 471](#)  
[\\_\\_init\\_\\_\(\) \(larray.ExcelReport method\), 409](#)  
[\\_\\_init\\_\\_\(\) \(larray.IGroup method\), 129](#)  
[\\_\\_init\\_\\_\(\) \(larray.LGroup method\), 138](#)  
[\\_\\_init\\_\\_\(\) \(larray.LSet method\), 148](#)  
[\\_\\_init\\_\\_\(\) \(larray.Metadata method\), 388](#)  
[\\_\\_init\\_\\_\(\) \(larray.ReportSheet method\), 414](#)  
[\\_\\_init\\_\\_\(\) \(larray.Session method\), 441](#)  
[\\_\\_init\\_\\_\(\) \(larray.Workbook method\), 406](#)  
[\\_\\_init\\_\\_\(\) \(larray.set\\_options method\), 426](#)

## A

[absolute\(\) \(in module larray\), 330](#)  
[add\(\) \(larray.Session method\), 453](#)  
[add\\_graph\(\) \(larray.ReportSheet method\), 418](#)  
[add\\_graphs\(\) \(larray.ReportSheet method\), 419](#)  
[add\\_title\(\) \(larray.ReportSheet method\), 417](#)  
[align\(\) \(larray.Array method\), 274](#)  
[align\(\) \(larray.Axis method\), 124](#)  
[align\(\) \(larray.AxisCollection method\), 170](#)  
[all\(\) \(larray.Array method\), 285](#)  
[all\\_by\(\) \(larray.Array method\), 287](#)  
[allclose\(\) \(larray.Array method\), 281](#)  
[angle\(\) \(in module larray\), 379](#)  
[any\(\) \(larray.Array method\), 289](#)  
[any\\_by\(\) \(larray.Array method\), 291](#)  
[app\(\) \(larray.Workbook method\), 407](#)  
[append\(\) \(larray.Array method\), 271](#)  
[append\(\) \(larray.AxisCollection method\), 160](#)  
[apply\(\) \(larray.Array method\), 201](#)  
[apply\(\) \(larray.Axis method\), 121](#)  
[apply\(\) \(larray.Session method\), 455](#)  
[apply\\_map\(\) \(larray.Array method\), 204](#)  
[arccos\(\) \(in module larray\), 363](#)  
[arccosh\(\) \(in module larray\), 377](#)  
[arcsin\(\) \(in module larray\), 362](#)

[arcsinh\(\) \(in module larray\), 376](#)  
[arctan\(\) \(in module larray\), 365](#)  
[arctan2\(\) \(in module larray\), 367](#)  
[arctanh\(\) \(in module larray\), 378](#)  
[Array \(class in larray\), 173](#)  
[arrays\(\) \(in module larray\), 443](#)  
[asarray\(\) \(in module larray\), 422](#)  
[astype\(\) \(larray.Array method\), 190](#)  
[astype\(\) \(larray.Axis method\), 125](#)  
[Axis \(class in larray\), 109](#)  
[axis\\_id\(\) \(larray.AxisCollection method\), 155](#)  
[AxisCollection \(class in larray\), 150](#)

## B

[broadcast\\_with\(\) \(larray.Array method\), 274](#)  
[by\(\) \(larray.Axis method\), 118](#)  
[by\(\) \(larray.IGroup method\), 131](#)  
[by\(\) \(larray.LGroup method\), 141](#)

## C

[ceil\(\) \(in module larray\), 343](#)  
[check\\_compatible\(\) \(larray.AxisCollection method\), 173](#)  
[CheckedArray\(\) \(in module larray\), 467](#)  
[CheckedParameters \(class in larray\), 473](#)  
[CheckedSession \(class in larray\), 467](#)  
[choice\(\) \(in module larray.random\), 485](#)  
[clip\(\) \(larray.Array method\), 310](#)  
[close\(\) \(larray.Workbook method\), 407](#)  
[combine\\_axes\(\) \(larray.Array method\), 209](#)  
[combine\\_axes\(\) \(larray.AxisCollection method\), 167](#)  
[compact\(\) \(larray.Array method\), 265](#)  
[compact\(\) \(larray.Session method\), 459](#)  
[compare\(\) \(in module larray\), 478](#)  
[conj\(\) \(in module larray\), 382](#)  
[containing\(\) \(larray.Axis method\), 114](#)  
[containing\(\) \(larray.IGroup method\), 134](#)  
[containing\(\) \(larray.LGroup method\), 144](#)  
[convolve\(\) \(in module larray\), 328](#)  
[copy\(\) \(larray.Array method\), 190](#)  
[copy\(\) \(larray.Axis method\), 112](#)  
[copy\(\) \(larray.AxisCollection method\), 154](#)

`copy()` (*larray.Session method*), 449  
`copysign()` (*in module larray*), 384  
`cos()` (*in module larray*), 359  
`cosh()` (*in module larray*), 374  
`cumprod()` (*larray.Array method*), 224  
`cumsum()` (*larray.Array method*), 223

## D

`debug()` (*in module larray*), 477  
`degrees()` (*in module larray*), 369  
`describe()` (*larray.Array method*), 256  
`describe_by()` (*larray.Array method*), 257  
`diag()` (*in module larray*), 431  
`diff()` (*larray.Array method*), 313  
`difference()` (*larray.Axis method*), 123  
`difference()` (*larray.IGroup method*), 134  
`difference()` (*larray.LGroup method*), 144  
`display_names` (*larray.AxisCollection property*), 152  
`divnot0()` (*larray.Array method*), 309  
`drop()` (*larray.Array method*), 199  
`dtype` (*larray.Array property*), 193  
`dump()` (*larray.Array method*), 403

## E

`e` (*in module larray.core.constants*), 488  
`edit()` (*in module larray*), 477  
`element_equals()` (*larray.Session method*), 449  
`empty()` (*in module larray*), 187  
`empty_like()` (*in module larray*), 188  
`endingwith()` (*larray.Axis method*), 114  
`endingwith()` (*larray.IGroup method*), 135  
`endingwith()` (*larray.LGroup method*), 145  
`eq()` (*larray.Array method*), 282  
`equals()` (*larray.Array method*), 278  
`equals()` (*larray.Axis method*), 127  
`equals()` (*larray.IGroup method*), 132  
`equals()` (*larray.LGroup method*), 141  
`equals()` (*larray.Session method*), 451  
`euler_gamma` (*in module larray.core.constants*), 488  
`ExcelReport` (*class in larray*), 408  
`exp()` (*in module larray*), 347  
`exp2()` (*in module larray*), 350  
`expand()` (*larray.Array method*), 269  
`expm1()` (*in module larray*), 349  
`extend()` (*larray.Array method*), 272  
`extend()` (*larray.Axis method*), 119  
`extend()` (*larray.AxisCollection method*), 160  
`eye()` (*in module larray*), 432

## F

`fabs()` (*in module larray*), 331  
`filter()` (*larray.Array method*), 201  
`filter()` (*larray.Session method*), 458

`fix()` (*in module larray*), 346  
`floor()` (*in module larray*), 342  
`frexp()` (*in module larray*), 385  
`from_frame()` (*in module larray*), 422  
`from_series()` (*in module larray*), 424  
`full()` (*in module larray*), 188  
`full_like()` (*in module larray*), 189

## G

`get()` (*larray.AxisCollection method*), 157  
`get()` (*larray.Session method*), 452  
`get_all()` (*larray.AxisCollection method*), 158  
`get_by_pos()` (*larray.AxisCollection method*), 158  
`get_example_filepath()` (*in module larray*), 425  
`get_options()` (*in module larray*), 427  
`global_arrays()` (*in module larray*), 444  
`graphs_per_row` (*larray.ExcelReport property*), 412  
`graphs_per_row` (*larray.ReportSheet property*), 417  
`growth_rate()` (*larray.Array method*), 255

## H

`hypot()` (*in module larray*), 366

## I

`i` (*larray.Array attribute*), 195  
`i` (*larray.Axis attribute*), 118  
`i0()` (*in module larray*), 338  
`identity()` (*in module larray*), 431  
`ids` (*larray.AxisCollection property*), 155  
`iflat` (*larray.Array attribute*), 197  
`ignore_labels()` (*larray.Array method*), 200  
`ignore_labels()` (*larray.Axis method*), 125  
`IGroup` (*class in larray*), 129  
`imag()` (*in module larray*), 381  
`index()` (*larray.Axis method*), 113  
`index()` (*larray.AxisCollection method*), 154  
`indexofmax()` (*larray.Array method*), 303  
`indexofmin()` (*larray.Array method*), 302  
`indicesofsorted()` (*larray.Array method*), 262  
`inf` (*in module larray.core.constants*), 488  
`info` (*larray.Array property*), 192  
`info` (*larray.AxisCollection property*), 153  
`insert()` (*larray.Array method*), 272  
`insert()` (*larray.Axis method*), 120  
`insert()` (*larray.AxisCollection method*), 161  
`interp()` (*in module larray*), 326  
`intersection()` (*larray.Axis method*), 122  
`intersection()` (*larray.IGroup method*), 133  
`intersection()` (*larray.LGroup method*), 143  
`inverse()` (*in module larray*), 325  
`ipfp()` (*in module larray*), 433  
`ipoints` (*larray.Array attribute*), 196  
`isaxis()` (*larray.AxisCollection method*), 172

[iscompatible\(\)](#) (*larray.Axis method*), 126  
[isin\(\)](#) (*larray.Array method*), 283  
[isinf\(\)](#) (*in module larray*), 334  
[isnan\(\)](#) (*in module larray*), 333  
[isscalar\(\)](#) (*in module larray*), 332  
[items\(\)](#) (*larray.Array method*), 307  
[items\(\)](#) (*larray.Session method*), 447  
[iter\\_labels\(\)](#) (*larray.AxisCollection method*), 156

## K

[keys\(\)](#) (*larray.Array method*), 304  
[keys\(\)](#) (*larray.AxisCollection method*), 154  
[keys\(\)](#) (*larray.Session method*), 446

## L

[labelofmax\(\)](#) (*larray.Array method*), 302  
[labelofmin\(\)](#) (*larray.Array method*), 301  
[labels](#) (*larray.AxisCollection property*), 152  
[labels\\_array\(\)](#) (*in module larray*), 427  
[labelsofsorted\(\)](#) (*larray.Array method*), 261  
[ldexp\(\)](#) (*in module larray*), 386  
[LGroup](#) (*class in larray*), 138  
[load\(\)](#) (*larray.Session method*), 460  
[load\\_example\\_data\(\)](#) (*in module larray*), 444  
[local\\_arrays\(\)](#) (*in module larray*), 443  
[log\(\)](#) (*in module larray*), 351  
[log10\(\)](#) (*in module larray*), 352  
[log1p\(\)](#) (*in module larray*), 354  
[log2\(\)](#) (*in module larray*), 353  
[logaddexp\(\)](#) (*in module larray*), 355  
[logaddexp2\(\)](#) (*in module larray*), 357  
[LSet](#) (*class in larray*), 148

## M

[matching\(\)](#) (*larray.Axis method*), 115  
[matching\(\)](#) (*larray.IGroup method*), 136  
[matching\(\)](#) (*larray.LGroup method*), 146  
[max\(\)](#) (*larray.Array method*), 297  
[max\(\)](#) (*larray.Axis method*), 116  
[max\\_by\(\)](#) (*larray.Array method*), 299  
[maximum\(\)](#) (*in module larray*), 322  
[mean\(\)](#) (*larray.Array method*), 225  
[mean\\_by\(\)](#) (*larray.Array method*), 227  
[median\(\)](#) (*larray.Array method*), 229  
[median\\_by\(\)](#) (*larray.Array method*), 231  
[memory\\_used](#) (*larray.Array property*), 194  
[Metadata](#) (*class in larray*), 387  
[min\(\)](#) (*larray.Array method*), 293  
[min\(\)](#) (*larray.Axis method*), 116  
[min\\_by\(\)](#) (*larray.Array method*), 295  
[minimum\(\)](#) (*in module larray*), 324

## N

[named\(\)](#) (*larray.IGroup method*), 130

[named\(\)](#) (*larray.LGroup method*), 140  
[names](#) (*larray.AxisCollection property*), 151  
[names](#) (*larray.Session property*), 445  
[nan](#) (*in module larray.core.constants*), 488  
[nan\\_to\\_num\(\)](#) (*in module larray*), 335  
[nbytes](#) (*larray.Array property*), 194  
[ndim](#) (*larray.Array property*), 193  
[ndtest\(\)](#) (*in module larray*), 182  
[new\\_sheet\(\)](#) (*larray.ExcelReport method*), 412  
[newline\(\)](#) (*larray.ReportSheet method*), 421  
[nonzero\(\)](#) (*larray.Array method*), 284  
[normal\(\)](#) (*in module larray.random*), 481

## O

[ones\(\)](#) (*in module larray*), 185  
[ones\\_like\(\)](#) (*in module larray*), 186  
[open\\_excel\(\)](#) (*in module larray*), 404

## P

[percent\(\)](#) (*larray.Array method*), 253  
[percentile\(\)](#) (*larray.Array method*), 241  
[percentile\\_by\(\)](#) (*larray.Array method*), 245  
[permutation\(\)](#) (*in module larray.random*), 485  
[pi](#) (*in module larray.core.constants*), 488  
[plot](#) (*larray.Array property*), 318  
[points](#) (*larray.Array attribute*), 196  
[pop\(\)](#) (*larray.AxisCollection method*), 159  
[prepend\(\)](#) (*larray.Array method*), 270  
[prod\(\)](#) (*larray.Array method*), 219  
[prod\\_by\(\)](#) (*larray.Array method*), 221  
[ptp\(\)](#) (*larray.Array method*), 250

## R

[radians\(\)](#) (*in module larray*), 370  
[randint\(\)](#) (*in module larray.random*), 479  
[ratio\(\)](#) (*larray.Array method*), 254  
[rationot0\(\)](#) (*larray.Array method*), 254  
[read\\_csv\(\)](#) (*in module larray*), 389  
[read\\_eurostat\(\)](#) (*in module larray*), 397  
[read\\_excel\(\)](#) (*in module larray*), 392  
[read\\_hdf\(\)](#) (*in module larray*), 396  
[read\\_sas\(\)](#) (*in module larray*), 397  
[read\\_stata\(\)](#) (*in module larray*), 398  
[read\\_tsv\(\)](#) (*in module larray*), 392  
[real\(\)](#) (*in module larray*), 380  
[reindex\(\)](#) (*larray.Array method*), 265  
[rename\(\)](#) (*larray.Array method*), 206  
[rename\(\)](#) (*larray.Axis method*), 119  
[rename\(\)](#) (*larray.AxisCollection method*), 162  
[replace\(\)](#) (*larray.Axis method*), 121  
[replace\(\)](#) (*larray.AxisCollection method*), 163  
[ReportSheet](#) (*class in larray*), 414  
[reshape\(\)](#) (*larray.Array method*), 263



`reshape_like()` (*larray.Array* method), 264  
`reverse()` (*larray.Array* method), 213  
`rint()` (*in module larray*), 345  
`roll()` (*larray.Array* method), 312  
`round()` (*in module larray*), 341  
`run_editor_on_exception()` (*in module larray*), 479

## S

`save()` (*larray.Session* method), 461  
`save()` (*larray.Workbook* method), 407  
`sequence()` (*in module larray*), 180  
*Session* (class *in larray*), 440  
`set()` (*larray.Array* method), 198  
`set_axes()` (*larray.Array* method), 205  
`set_item_default_size()` (*larray.ExcelReport* method), 411  
`set_item_default_size()` (*larray.ReportSheet* method), 416  
`set_labels()` (*larray.Array* method), 207  
`set_labels()` (*larray.AxisCollection* method), 164  
`set_options` (class *in larray*), 425  
`shape` (*larray.Array* property), 192  
`shape` (*larray.AxisCollection* property), 152  
`sheet_names()` (*larray.ExcelReport* method), 413  
`sheet_names()` (*larray.Workbook* method), 406  
`shift()` (*larray.Array* method), 311  
`signbit()` (*in module larray*), 383  
`sin()` (*in module larray*), 358  
`sinc()` (*in module larray*), 339  
`sinh()` (*in module larray*), 372  
`size` (*larray.Array* property), 193  
`size` (*larray.AxisCollection* property), 153  
`sort_labels()` (*larray.Array* method), 259  
`sort_values()` (*larray.Array* method), 260  
`split()` (*larray.Axis* method), 124  
`split_axes()` (*larray.Array* method), 211  
`split_axes()` (*larray.AxisCollection* method), 169  
`sqrt()` (*in module larray*), 337  
`stack()` (*in module larray*), 428  
`startingwith()` (*larray.Axis* method), 114  
`startingwith()` (*larray.IGroup* method), 135  
`startingwith()` (*larray.LGroup* method), 145  
`std()` (*larray.Array* method), 237  
`std_by()` (*larray.Array* method), 239  
`sum()` (*larray.Array* method), 214  
`sum_by()` (*larray.Array* method), 217  
`summary()` (*larray.Session* method), 447

## T

`tan()` (*in module larray*), 361  
`tanh()` (*in module larray*), 375  
`template` (*larray.ExcelReport* property), 411  
`template` (*larray.ReportSheet* property), 416  
`template_dir` (*larray.ExcelReport* property), 410

`template_dir` (*larray.ReportSheet* property), 415  
`to_clipboard()` (*larray.Array* method), 315  
`to_csv()` (*larray.Array* method), 399  
`to_csv()` (*larray.Session* method), 463  
`to_excel()` (*larray.Array* method), 400  
`to_excel()` (*larray.ExcelReport* method), 413  
`to_excel()` (*larray.Session* method), 464  
`to_frame()` (*larray.Array* method), 317  
`to_hdf()` (*larray.Array* method), 401  
`to_hdf()` (*larray.Axis* method), 128  
`to_hdf()` (*larray.IGroup* method), 137  
`to_hdf()` (*larray.LGroup* method), 147  
`to_hdf()` (*larray.Session* method), 465  
`to_pickle()` (*larray.Session* method), 466  
`to_series()` (*larray.Array* method), 315  
`to_stata()` (*larray.Array* method), 402  
`translate()` (*larray.IGroup* method), 133  
`translate()` (*larray.LGroup* method), 142  
`transpose()` (*larray.Array* method), 268  
`transpose()` (*larray.Session* method), 457  
`trunc()` (*in module larray*), 344

## U

`uniform()` (*in module larray.random*), 483  
`union()` (*in module larray*), 428  
`union()` (*larray.Axis* method), 122  
`union()` (*larray.IGroup* method), 133  
`union()` (*larray.LGroup* method), 142  
`unique()` (*larray.Array* method), 314  
`unwrap()` (*in module larray*), 371  
`update()` (*larray.Session* method), 453

## V

`value_counts()` (*larray.Array* method), 258  
`values()` (*larray.Array* method), 305  
`values()` (*larray.Session* method), 446  
`var()` (*larray.Array* method), 233  
`var_by()` (*larray.Array* method), 235  
`view()` (*in module larray*), 476

## W

`where()` (*in module larray*), 321  
`with_axis()` (*larray.IGroup* method), 131  
`with_axis()` (*larray.LGroup* method), 140  
`with_total()` (*larray.Array* method), 252  
`without()` (*larray.AxisCollection* method), 166  
*Workbook* (class *in larray*), 406  
`wrap_elementwise_array_func()` (*in module larray*), 435

## Z

`zeros()` (*in module larray*), 184  
`zeros_like()` (*in module larray*), 185



`zip_array_items()` (*in module larray*), [438](#)  
`zip_array_values()` (*in module larray*), [436](#)