xnumpy Documentation

Release 0.1.3+0.ga6f77e4.dirty

John Kirkham

Nov 08, 2018

Contents

Contents:

CHAPTER 1

xnumpy

A Python utility library for working with expanded NumPy arrays.

- Free software: BSD 3-Clause
- Documentation: https://xnumpy.readthedocs.io.

1.1 Features

- TODO

1.2 Credits

This package was created with Cookiecutter and the nanshe-org/nanshe-cookiecutter project template.

CHAPTER 2

Installation

2.1 Stable release

To install xnumpy, run this command in your terminal:

```
$ pip install xnumpy
```

This is the preferred method to install xnumpy, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for xnumpy can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/jakirkham/xnumpy
```

Or download the tarball:

```
$ curl  -OL https://github.com/jakirkham/xnumpy/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use xnumpy in a project:

```
import xnumpy
```

CHAPTER 4

API

4.1 xnumpy package

4.1.1 Submodules

xnumpy.core module

xnumpy.core.**anumerate**(*new_array*, *axis=0*, *start=0*, *step=1*, *dtype=None*)

Builds on expand_arange, which has the same shape as the original array. Specifies the increments to occur along the given axis, which by default is the zeroth axis. Provides mechanisms for changing the starting value and also the increment. :param new_array: array to enumerate :type new_array: numpy.ndarray :param start: starting point (0 by default). :type start: int :param step: size of steps to take between

value (1 by default).

Parameters

- **axis** (*int*) – axis to enumerate along (0 by default).
- **dtype** (*type*) – type to use for enumerating.

Returns

a view of a numpy arange with tiling in various dimension.

Return type (numpy.ndarray)

Examples

```
>>> a = numpy.ones((4,5), dtype=numpy.uint64)
```

```
>>> anumerate(a)
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3]], dtype=uint64)
```

```
>>> anumerate(a, axis=0)
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3]], dtype=uint64)
```

```
>>> anumerate(a, axis=0, start=1)
array([[1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]], dtype=uint64)
```

```
>>> anumerate(a, axis=0, start=1, step=2)
array([[1, 1, 1, 1, 1],
       [3, 3, 3, 3, 3],
       [5, 5, 5, 5, 5],
       [7, 7, 7, 7, 7]], dtype=uint64)
```

```
>>> anumerate(a, axis=1)
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]], dtype=uint64)
```

```
>>> anumerate(a, axis=1, start=1)
array([[1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5]], dtype=uint64)
```

```
>>> anumerate(a, axis=1, start=1, step=2)
array([[1, 3, 5, 7, 9],
       [1, 3, 5, 7, 9],
       [1, 3, 5, 7, 9],
       [1, 3, 5, 7, 9]], dtype=uint64)
```

xnumpy.core.**expand**(*new_array*, *shape_after=()*, *shape_before=()*, *read_only=True*)

Tack on extra dimensions of the specified size to either side.

Behaves like NumPy tile except that it always returns a view not a copy. Though, it differs in that additional dimensions are added for repetition as opposed to repeating in the same one. Also, it allows repetitions to be specified before or after unlike tile. Though, will behave identical to tile if the keyword is not specified. Uses strides to trick NumPy into providing a view.

Parameters

- **new_array** (*numpy.ndarray*) – array to expand
- **shape_after** (*tuple or int*) – size of dimensions to add before the array shape (if int will turn into tuple). Defaults to an empty tuple.

- **shape_before** (*tuple or int*) – size of dimensions to add before the array shape (if int will turn into tuple). Defaults to an empty tuple.
- **read_only** (*bool*) –

Returns

a view of a numpy array with tiling in various dimension.

Return type (numpy.ndarray)

Examples

```
>>> numpy.set_printoptions(legacy="1.13")
```

```
>>> a = numpy.arange(6, dtype=numpy.uint64).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]], dtype=uint64)
```

```
>>> expand(a)
array([[0, 1, 2],
       [3, 4, 5]], dtype=uint64)
```

```
>>> a is expand(a)
False
```

```
>>> expand(a, 1)
array([[[0],
        [1],
        [2]],
<BLANKLINE>
       [[3],
        [4],
        [5]]], dtype=uint64)
```

```
>>> expand(a, (1,))
array([[[0],
        [1],
        [2]],
<BLANKLINE>
       [[3],
        [4],
        [5]]], dtype=uint64)
```

```
>>> expand(a, shape_after=1)
array([[[0],
        [1],
        [2]],
<BLANKLINE>
       [[3],
        [4],
        [5]]], dtype=uint64)
```

```
>>> expand(a, shape_after=(1,))
array([[[0],
        [1],
        [2]],
<BLANKLINE>
       [[3],
        [4],
        [5]]], dtype=uint64)
```

```
>>> expand(a, shape_before=1)
array([[[0, 1, 2],
        [3, 4, 5]]], dtype=uint64)
```

```
>>> expand(a, shape_before=(1,))
array([[[0, 1, 2],
        [3, 4, 5]]], dtype=uint64)
```

```
>>> expand(a, shape_before=(3,))
array([[[0, 1, 2],
        [3, 4, 5]],
<BLANKLINE>
       [[0, 1, 2],
        [3, 4, 5]],
<BLANKLINE>
       [[0, 1, 2],
        [3, 4, 5]]], dtype=uint64)
```

```
>>> expand(a, shape_after=(4,))
array([[[0, 0, 0, 0],
        [1, 1, 1, 1],
        [2, 2, 2, 2]],
<BLANKLINE>
       [[3, 3, 3, 3],
        [4, 4, 4, 4],
        [5, 5, 5, 5]]], dtype=uint64)
```

```
>>> expand(
...     a,
...     shape_before=(3,),
...     shape_after=(4,)
... )
array([[[[0, 0, 0, 0],
         [1, 1, 1, 1],
         [2, 2, 2, 2]],
<BLANKLINE>
        [[3, 3, 3, 3],
         [4, 4, 4, 4],
         [5, 5, 5, 5]]],
<BLANKLINE>
<BLANKLINE>
       [[[0, 0, 0, 0],
         [1, 1, 1, 1],
         [2, 2, 2, 2]],
<BLANKLINE>
        [[3, 3, 3, 3],
```

(continues on next page)

(continued from previous page)

```
         [4, 4, 4, 4],
         [5, 5, 5, 5]]],
<BLANKLINE>
<BLANKLINE>
       [[[0, 0, 0, 0],
         [1, 1, 1, 1],
         [2, 2, 2, 2]],
<BLANKLINE>
        [[3, 3, 3, 3],
         [4, 4, 4, 4],
         [5, 5, 5, 5]]]], dtype=uint64)
```

```
>>> expand(a, shape_after=(4,3))
array([[[[0, 0, 0],
         [0, 0, 0],
         [0, 0, 0],
         [0, 0, 0]],
<BLANKLINE>
        [[1, 1, 1],
         [1, 1, 1],
         [1, 1, 1],
         [1, 1, 1]],
<BLANKLINE>
        [[2, 2, 2],
         [2, 2, 2],
         [2, 2, 2],
         [2, 2, 2]]],
<BLANKLINE>
<BLANKLINE>
       [[[3, 3, 3],
         [3, 3, 3],
         [3, 3, 3],
         [3, 3, 3]],
<BLANKLINE>
        [[4, 4, 4],
         [4, 4, 4],
         [4, 4, 4],
         [4, 4, 4]],
<BLANKLINE>
        [[5, 5, 5],
         [5, 5, 5],
         [5, 5, 5],
         [5, 5, 5]]]], dtype=uint64)
```

```
>>> expand(
...     a,
...     shape_before=(4,3),
... )
array([[[[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]]],
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
<BLANKLINE>
       [[[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]]],
<BLANKLINE>
<BLANKLINE>
       [[[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]]],
<BLANKLINE>
<BLANKLINE>
       [[[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]],
<BLANKLINE>
        [[0, 1, 2],
         [3, 4, 5]]]], dtype=uint64)
```

xnumpy.core.**indices**(*shape*, *dtype=None*)

Provides a function similar to numpy.indices.

Very similar to numpy.indices except for the following.

- Only returns a tuple of the index arrays for each dimension.
- Returns an expanded view of a ‘‘numpy.arange‘‘s.

This result in drastic improvements in performance in terms of time, memory usage, and cache retrieval.

To get an equivalent result to numpy.indices, one merely need to call numpy.array(indices(shape, dtype=int)).

Parameters

- **shape** (tuple of ‘‘int‘‘s) – array to enumerate
- **dtype** (type) – type to use for enumerating.

Returns a tuple of index arrays.

Return type tuple of ‘‘numpy.ndarray‘‘s

Examples

```
>>> indices(
...     (2, 3),
...     dtype=numpy.uint64
... )  # doctest: +NORMALIZE_WHITESPACE
(array([[0, 0, 0],
        [1, 1, 1]], dtype=uint64),
 array([[0, 1, 2],
        [0, 1, 2]], dtype=uint64))
```

xnumpy.core.**permute_op**(*op*, *array_1*, *array_2*)

Takes two arrays and constructs a new array that contains the result of new_op on every permutation of elements in each array (like broadcasting).

Suppose that new_result contained the result, then one would find that the result of the following operation on the specific indices.

new_op(array_1[i_1_1, i_1_2, . . .], array_2[i_2_1, i_2_2, . . .])

would be found in new_result as shown

new_result[i_1_1, i_1_2, . . . , i_2_1, i_2_2, . . .]

Parameters **new_array** (*numpy.ndarray*) – array to add the singleton axis to.

Returns

a numpy array with the singleton axis added at the end.

Return type (numpy.ndarray)

Examples

```
>>> import operator
>>> tuple(int(s) for s in permute_op(
...     operator.add, numpy.ones((1,3)), numpy.eye(2)
... ).shape)
(1, 3, 2, 2)
```

```
>>> tuple(int(s) for s in permute_op(
...     operator.add, numpy.ones((2,2)), numpy.eye(2)
... ).shape)
(2, 2, 2, 2)
```

```
>>> permute_op(
...     operator.add, numpy.ones((2,2)), numpy.eye(2)
... )
array([[[[ 2.,  1.],
         [ 1.,  2.]],
<BLANKLINE>
        [[ 2.,  1.],
         [ 1.,  2.]]],
<BLANKLINE>
<BLANKLINE>
       [[[ 2.,  1.],
         [ 1.,  2.]],
<BLANKLINE>
        [[ 2.,  1.],
         [ 1.,  2.]]]])
```

```
>>> permute_op(
...     operator.sub, numpy.ones((2,2)), numpy.eye(2)
... )
array([[[[ 0.,  1.],
         [ 1.,  0.]],
<BLANKLINE>
        [[ 0.,  1.],
         [ 1.,  0.]]],
<BLANKLINE>
<BLANKLINE>
       [[[ 0.,  1.],
         [ 1.,  0.]],
<BLANKLINE>
        [[ 0.,  1.],
         [ 1.,  0.]]]])
```

```
>>> permute_op(
...     operator.sub, numpy.ones((2,2)), numpy.fliplr(numpy.eye(2))
... )
array([[[[ 1.,  0.],
         [ 0.,  1.]],
<BLANKLINE>
        [[ 1.,  0.],
         [ 0.,  1.]]],
<BLANKLINE>
<BLANKLINE>
       [[[ 1.,  0.],
         [ 0.,  1.]],
<BLANKLINE>
        [[ 1.,  0.],
         [ 0.,  1.]]]])
```

```
>>> permute_op(
...     operator.sub, numpy.zeros((2,2)), numpy.eye(2)
... )
array([[[[-1.,  0.],
         [ 0., -1.]],
<BLANKLINE>
        [[-1.,  0.],
         [ 0., -1.]]],
<BLANKLINE>
<BLANKLINE>
       [[[-1.,  0.],
         [ 0., -1.]],
<BLANKLINE>
        [[-1.,  0.],
         [ 0., -1.]]]])
```

xnumpy.core.**product**(*arrays*)

Takes the cartesian product between the elements in each array.

Parameters **arrays** (*collections.Sequence of numpy.ndarrays*) – A sequence of 1-D arrays or a 2-D array.

Returns

an array containing the result of the cartesian product of each array.

Return type (numpy.ndarray)

Examples

```
>>> product([
...     numpy.arange(2, dtype=numpy.uint64),
...     numpy.arange(3, dtype=numpy.uint64)
... ])
array([[0, 0],
       [0, 1],
       [0, 2],
       [1, 0],
       [1, 1],
       [1, 2]], dtype=uint64)
```

```
>>> product([
...     numpy.arange(2, dtype=float),
...     numpy.arange(3, dtype=numpy.uint64)
... ])
array([[ 0.,  0.],
       [ 0.,  1.],
       [ 0.,  2.],
       [ 1.,  0.],
       [ 1.,  1.],
       [ 1.,  2.]])
```

```
>>> product([
...     numpy.arange(2, dtype=numpy.uint64),
...     numpy.arange(3, dtype=numpy.uint64),
...     numpy.arange(4, dtype=numpy.uint64)
... ])
array([[0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 0, 3],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2],
       [0, 1, 3],
       [0, 2, 0],
       [0, 2, 1],
       [0, 2, 2],
       [0, 2, 3],
       [1, 0, 0],
       [1, 0, 1],
       [1, 0, 2],
       [1, 0, 3],
       [1, 1, 0],
       [1, 1, 1],
       [1, 1, 2],
       [1, 1, 3],
       [1, 2, 0],
       [1, 2, 1],
       [1, 2, 2],
       [1, 2, 3]], dtype=uint64)
```

```
>>> product(numpy.diag((1, 2, 3)).astype(numpy.uint64))
array([[1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
       [1, 0, 0],
       [1, 0, 3],
       [1, 2, 0],
       [1, 2, 0],
       [1, 2, 3],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 3],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 3],
       [0, 2, 0],
       [0, 2, 0],
       [0, 2, 3],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 3],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 3],
       [0, 2, 0],
       [0, 2, 0],
       [0, 2, 3],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 3]], dtype=uint64)
```

CHAPTER 5

Contributing

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

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/jakirkham/xnumpy/issues.

If you are reporting a bug, please include:

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

5.1.2 Fix Bugs

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

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

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

5.1.5 Submit Feedback

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

If you are proposing a feature:

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

5.2 Get Started!

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

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

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

3. Install your local copy into an environment. Assuming you have conda installed, this is how you set up your fork for local development (on Windows drop *source*). Replace *"<some version>"* with the Python version used for testing.:

```
$ conda create -n xnumpyenv python="<some version>"
$ source activate xnumpyenv
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

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

```
$ flake8 xnumpy tests
$ python setup.py test or py.test
```

   To get flake8, just conda install it into your environment.

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

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

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5, and 3.6. Check https://travis-ci.org/jakirkham/xnumpy/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

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

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

x

Index