dask Documentation

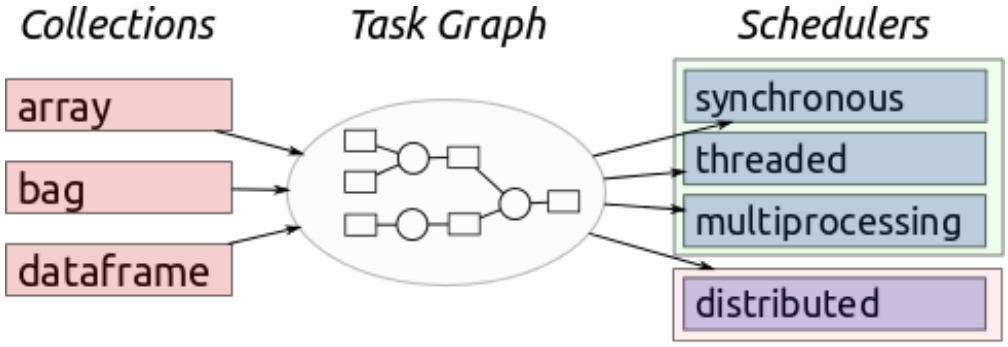
Release 0.10.0

Dask Development Team

June 27, 2016

Dask is a flexible parallel computing library for analytics. Dask emphasizes the following virtues:

- **Familiar**: Provides parallelized NumPy array and Pandas DataFrame objects
- **Native**: Enables distributed computing in Pure Python with access to the PyData stack.
- **Fast**: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms
- **Flexible**: Supports complex and messy workloads
- **Scales up**: Runs resiliently on clusters with 100s of nodes
- **Scales down**: Trivial to set up and run on a laptop in a single process
- **Responsive**: Designed with interactive computing in mind it provides rapid feedback and diagnostics to aid humans



CHAPTER 1

Familiar user interface

Dask DataFrame mimics Pandas

```
import pandas as pd                          import dask.dataframe as dd
df = pd.read_csv('2015-01-01.csv')           df = dd.read_csv('2015-*-*.csv')
df.groupby(df.user_id).value.mean()          df.groupby(df.user_id).value.mean().compute()
```

Dask Array mimics NumPy

```
import numpy as np                           import dask.array as da
f = h5py.File('myfile.hdf5')                 f = h5py.File('myfile.hdf5')
x = np.array(f['/small-data'])               x = da.from_array(f['/big-data'],
                                                               chunks=(1000, 1000))
x - x.mean(axis=1)                           x - x.mean(axis=1).compute()
```

Dask Bag mimics iterators, Toolz, PySpark

```
import dask.bag as db
b = db.read_text('2015-*-*.json.gz').map(json.loads)
b.pluck('name').frequencies().topk(10, lambda pair: pair[1]).compute()
```

Dask Delayed mimics for loops and wraps custom code

```
from dask import delayed
L = []
for fn in filenames:                    # Use for loops to build up computation
    data = delayed(load)(fn)            # Delay execution of function
    L.append(delayed(process)(data))    # Build connections between variables

result = delayed(summarize)(L)
result.compute()
```

CHAPTER 2

Scales from laptops to clusters

Dask is convenient on a laptop. It installs trivially with `conda` or `pip` and extends the size of convenient datasets from "fits in memory" to "fits on disk".

Dask can scale to a cluster of 100s of machines. It is resilient, elastic, data local, and low latency. For more information see documentation on the distributed scheduler.

This ease of transition between single-machine to moderate cluster enables users both to start simple and to grow when necessary.

CHAPTER 3

Complex Algorithms

Dask represents parallel computations with task graphs. These directed acyclic graphs may have arbitrary structure, which enables both developers and users the freedom to build sophisticated algorithms and to handle messy situations not easily managed by the `map/filter/groupby` paradigm common in most data engineering frameworks.

We originally needed this complexity to build complex algorithms for n-dimensional arrays but have found it to be equally valuable when dealing with messy situations in everyday problems.

CHAPTER 4

Index

Getting Started

- Install Dask
- Examples and Tutorials
- Dask Cheat Sheet

4.1 Install Dask

You can install dask with `conda`, with `pip`, or by installing from source.

4.1.1 Conda

To install the latest version of Dask from the conda-forge repository using conda:

```
conda install dask -c conda-forge
```

This installs dask and all common dependencies, including Pandas and NumPy.

4.1.2 Pip

To install Dask with `pip` there are a few options, depending on which dependencies you would like to keep up to date:

- `pip install dask[complete]`: Install everything
- `pip install dask[array]`: Install dask and numpy
- `pip install dask[bag]`: Install dask and cloudpickle
- `pip install dask[dataframe]`: Install dask, numpy, and pandas
- `pip install dask`: Install only dask, which depends only on the standard library. This is appropriate if you only want the task schedulers.

We do this so that users of the lightweight core dask scheduler aren't required to download the more exotic dependencies of the collections (numpy, pandas, etc..)

4.1.3 Install from Source

To install dask from source, clone the repository from github:

```
git clone https://github.com/dask/dask.git
cd dask
python setup.py install
```

or use `pip` locally if you want to install all dependencies as well:

```
pip install -e .[complete]
```

You can view the list of all dependencies within the `extras_require` field of `setup.py`.

4.1.4 Test

Test dask with `py.test`:

```
cd dask
py.test dask
```

Although please aware that installing dask naively may not install all requirements by default. Please read the `pip` section above that discusses requirements. You may choose to install the `dask[complete]` which includes all dependencies for all collections. Alternatively you may choose to test only certain submodules depending on the libraries within your environment. For example to test only dask core and dask array we would run tests as follows:

```
py.test dask/tests dask/array/tests
```

4.2 Examples and Tutorials

4.2.1 Examples

You can view examples for each Dask collection:

- Dask Array Examples
- Dask Bag Examples
- Dask DataFrame Examples
- Dask delayed Examples

4.2.2 Tutorials

You can view a tutorial of Dask, including slides and notebooks on Dask arrays, graphs, dataframes, delayed, and bags in the dask-tutorials repository.

You can try a live tutorial of dask dataframe's basics here.

Dask dataframe's timeseries functionality is demonstrated on Binder. And static notebook with more data and profiling is on nbviewer.

4.2.3 Notebooks

You can view example Dask notebooks in the dask-examples repository.

4.3 Dask Cheat Sheet

The 200KB pdf `dask cheat sheet` is a single page summary of all the most important information about using dask.

Collections

Dask collections are the main interaction point for users. They look like NumPy and pandas but generate dask graphs internally. If you are a dask *user* then you should start here.

- Array
- Bag
- DataFrame
- Delayed

4.4 Array

Dask arrays implement a subset of the NumPy interface on large arrays using blocked algorithms and task scheduling.

4.4.1 Overview

Dask Array implements a subset of the NumPy ndarray interface using blocked algorithms, cutting up the large array into many small arrays. This lets us compute on arrays larger than memory using all of our cores. We coordinate these blocked algorithms using dask graphs.

Design

Dask arrays coordinate many NumPy arrays arranged into a grid. These NumPy arrays may live on disk or on other machines.

Common Uses

Today Dask array is commonly used in the sort of gridded data analysis that arises in weather, climate modeling, or oceanograhy, especially when data sizes become inconveniently large. Dask array complements large on-disk array stores like HDF5, NetCDF, and BColz. Additionally Dask.array is commonly used to speed up expensive in-memory computations using multiple cores, such as you might find in image analysis or statistical and machine learning applications.

Scope

The `dask.array` library supports the following interface from `numpy`:

- Arithmetic and scalar mathematics, `+, *, exp, log, ...`
- Reductions along axes, `sum(), mean(), std(), sum(axis=0), ...`
- Tensor contractions / dot products / matrix multiply, `tensordot`
- Axis reordering / transpose, `transpose`

- Slicing, `x[:100, 500:100:-2]`
- Fancy indexing along single axes with lists or numpy arrays, `x[:, [10, 1, 5]]`
- The array protocol `__array__`
- Some linear algebra `svd, qr, solve, solve_triangular, lstsq`

See the dask.array API for a more extensive list of functionality.

Execution

By default dask.array uses the threaded scheduler in order to avoid data transfer costs and because NumPy releases the GIL well. It is also quite effective on a cluster using the dask.distributed scheduler.

Limitations

Dask.array does not implement the entire numpy interface. Users expecting this will be disappointed. Notably, dask.array has the following limitations:

1. Dask does not implement all of `np.linalg`. This has been done by a number of excellent BLAS/LAPACK implementations, and is the focus of numerous ongoing academic research projects.
2. Dask.array does not support any operation where the resulting shape depends on the values of the array. In order to form the dask graph we must be able to infer the shape of the array before actually executing the operation. This precludes operations like indexing one dask array with another or operations like `np.where`.
3. Dask.array does not attempt operations like `sort` which are notoriously difficult to do in parallel, and are of somewhat diminished value on very large data (you rarely actually need a full sort). Often we include parallel-friendly alternatives like `topk`.
4. Dask development is driven by immediate need, and so many lesser used functions have not been implemented. Community contributions are encouraged.

4.4.2 Create Dask Arrays

We store and manipulate large arrays in a wide variety of ways. There are some standards like HDF5 and NetCDF but just as often people use custom storage solutions. This page talks about how to build dask graphs to interact with your array.

In principle we need functions that return NumPy arrays. These functions and their arrangement can be as simple or as complex as the situation dictates.

Simple case - Format Supports NumPy Slicing

Many storage formats have Python projects that expose storage using NumPy slicing syntax. These include HDF5, NetCDF, BColz, Zarr, GRIB, etc.. For example the `HDF5` file format has the `h5py` Python project, which provides a `Dataset` object into which we can slice in NumPy fashion.

```
>>> import h5py
>>> f = h5py.File('myfile.hdf5') # HDF5 file
>>> d = f['/data/path']          # Pointer on on-disk array
>>> d.shape                      # d can be very large
(1000000, 1000000)

>>> x = d[:5, :5]                # We slice to get numpy arrays
```

It is common for Python wrappers of on-disk array formats to present a NumPy slicing syntax. The full dataset looks like a NumPy array with `.shape` and `.dtype` attributes even though the data hasn't yet been loaded in and still lives on disk. Slicing in to this array-like object fetches the appropriate data from disk and returns that region as an in-memory NumPy array.

For this common case `dask.array` presents the convenience function `da.from_array`

```
>>> import dask.array as da
>>> x = da.from_array(d, chunks=(1000, 1000))
```

Concatenation and Stacking

Often we store data in several different locations and want to stitch them together.

```
>>> filenames = sorted(glob('2015-*-*.hdf5')
>>> dsets = [h5py.File(fn)['/data'] for fn in filenames]
>>> arrays = [da.from_array(dset, chunks=(1000, 1000)) for dset in dsets]
>>> x = da.concatenate(arrays, axis=0)  # Concatenate arrays along first axis
```

For more information see concatenation and stacking docs.

Using `dask.delayed`

You can create a plan to arrange many numpy arrays into a grid with normal for loops using dask.delayed and then convert these into a dask array later. See documentation on using dask.delayed with collections.

Raw dask graphs

If your format does not provide a convenient slicing solution you can dive down one layer to interact with dask dictionaries. Your goal is to create a dictionary with tasks that create NumPy arrays, see docs on array design before continuing with this subsection.

To construct a dask array manually you need a dict with tasks that form numpy arrays

```
dsk = {('x', 0): (f, ...),
       ('x', 1), (f, ...),
       ('x', 2): (f, ...)}
```

And a chunks tuple that defines the shapes of your blocks along each dimension

```
chunks = [(1000, 1000, 1000)]
```

For the tasks `(f, ...)` your choice of function `f` and arguments `...` is up to you. You have the full freedom of the Python language here as long as your function, when run with those arguments, produces the appropriate NumPy array.

Chunks

We always specify a `chunks` argument to tell dask.array how to break up the underlying array into chunks. This strongly impacts performance. We can specify `chunks` in one of three ways

- a blocksize like `1000`
- a blockshape like `(1000, 1000)`
- explicit sizes of all blocks along all dimensions, like `((1000, 1000, 500), (400, 400))`

Your chunks input will be normalized and stored in the third and most explicit form.

A good choice of `chunks` follows the following rules:

1. A chunk should be small enough to fit comfortably in memory. We'll have many chunks in memory at once.
2. A chunk must be large enough so that computations on that chunk take significantly longer than the 1ms overhead per task that dask scheduling incurs. A task should take longer than 100ms.
3. Chunks should align with the computation that you want to do. For example if you plan to frequently slice along a particular dimension then it's more efficient if your chunks are aligned so that you have to touch fewer chunks. If you want to add two arrays then its convenient if those arrays have matching chunks patterns.

Example

As an example, we might load a grid of pickle files known to contain 1000 by 1000 NumPy arrays.

```
def load(fn):
    with open(fn) as f:
        result = pickle.load(f)
     return result

dsk = {('x', 0, 0): (load, 'block-0-0.pkl'),
       ('x', 0, 1): (load, 'block-0-1.pkl'),
       ('x', 0, 2): (load, 'block-0-2.pkl'),
       ('x', 1, 0): (load, 'block-1-0.pkl'),
       ('x', 1, 1): (load, 'block-1-1.pkl'),
       ('x', 1, 2): (load, 'block-1-2.pkl')}

 chunks = ((1000, 1000), (1000, 1000, 1000))

 x = da.Array(dsk, 'x', chunks)
```

4.4.3 Store Dask Arrays

In Memory

If you have a small amount of data, you can call `np.array` on your dask array to turn in to a normal NumPy array:

```
>>> x = da.arange(6, chunks=3)
>>> y = x**2
>>> np.array(y)
array([0, 1, 4, 9, 16, 25])
```

HDF5

Use the `to_hdf5` function to store data into HDF5 using `h5py`:

```
>>> da.to_hdf5('myfile.hdf5', '/y', y)  # doctest: +SKIP
```

Store several arrays in one computation with the function `da.to_hdf5` by passing in a dict:

```
>>> da.to_hdf5('myfile.hdf5', {'/x': x, '/y': y})  # doctest: +SKIP
```

Other On-Disk Storage

Alternatively, you can store dask arrays in any object that supports numpy-style slice assignment like `h5py.Dataset`, or `bcolz.carray`:

```
>>> import bcolz  # doctest: +SKIP
>>> out = bcolz.zeros(shape=y.shape, rootdir='myfile.bcolz')  # doctest: +SKIP
>>> da.store(y, out)  # doctest: +SKIP
```

You can store several arrays in one computation by passing lists of sources and destinations:

```
>>> da.store([array1, array2], [output1, outpu2])
```

On-Disk Storage

In the example above we used `h5py`, but `dask.array` works equally well with `pytables`, `bcolz`, or any library that provides an array object from which we can slice out numpy arrays:

```
>>> x = dataset[1000:2000, :2000]  # pull out numpy array from on-disk object
```

This API has become a standard in the numeric Python ecosystem. Dask works with any object that supports this operation and the equivalent assignment syntax:

```
>>> dataset[1000:2000, :2000] = x  # Store numpy array in on-disk object
```

4.4.4 API

Top level user functions:

all(a[, axis, keepdims, split_every])	Test whether all array elements along a given axis evaluate to True.
angle	Return the angle of the complex argument.
any(a[, axis, keepdims, split_every])	Test whether any array element along a given axis evaluates to True.
arange(*args, **kwargs)	Return evenly spaced values from *start* to *stop* with step size *step*.
arccos(x[, out])	Trigonometric inverse cosine, element-wise.
arccosh(x[, out])	Inverse hyperbolic cosine, element-wise.
arcsin(x[, out])	Inverse sine, element-wise.
arcsinh(x[, out])	Inverse hyperbolic sine element-wise.
arctan(x[, out])	Trigonometric inverse tangent, element-wise.
arctan2(x1, x2[, out])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
arctanh(x[, out])	Inverse hyperbolic tangent element-wise.
argmax(x[, axis, split_every])	Indices of the maximum values along an axis.
argmin(x[, axis, split_every])	Return the indices of the minimum values along an axis.
around(x[, decimals])	Evenly round to the given number of decimals.
array(object[, dtype, copy, order, subok, ndmin])	Create an array.
bincount(x[, weights, minlength])	Count number of occurrences of each value in array of non-negative ints.
broadcast_to(x, shape)	Broadcast an array to a new shape.
coarsen(reduction, x, axes[, trim_excess])	Coarsen array by applying reduction to fixed size neighborhoods
ceil(x[, out])	Return the ceiling of the input, element-wise.
choose(a, choices)	Construct an array from an index array and a set of arrays to choose from.
clip	Clip (limit) the values in an array.
compress(condition, a[, axis])	Return selected slices of an array along given axis.
concatenate(seq[, axis])	Concatenate arrays along an existing axis

Table 4.1 – continued from previous page

conj(x[, out])	Return the complex conjugate, element-wise.
copysign(x1, x2[, out])	Change the sign of x1 to that of x2, element-wise.
corrcoef(x[, y, rowvar])	Return correlation coefficients.
cos(x[, out])	Cosine element-wise.
cosh(x[, out])	Hyperbolic cosine, element-wise.
cov(m[, y, rowvar, bias, ddof])	Estimate a covariance matrix, given data.
cumprod(x, axis[, dtype])	Return the cumulative product of elements along a given axis.
cumsum(x, axis[, dtype])	Return the cumulative sum of the elements along a given axis.
deg2rad(x[, out])	Convert angles from degrees to radians.
degrees(x[, out])	Convert angles from radians to degrees.
diag(v)	Extract a diagonal or construct a diagonal array.
dot(a, b[, out])	Dot product of two arrays.
dstack(tup)	Stack arrays in sequence depth wise (along third axis).
empty	Blocked variant of empty
exp(x[, out])	Calculate the exponential of all elements in the input array.
expm1(x[, out])	Calculate `exp(x) - 1` for all elements in the array.
eye(N, chunks[, M, k, dtype])	Return a 2-D Array with ones on the diagonal and zeros elsewhere.
fabs(x[, out])	Compute the absolute values element-wise.
fft.fft(a[, n, axis])	Wrapping of numpy.fft.fft
fft.ifft(a[, n, axis])	Wrapping of numpy.fft.ifft
fft.hfft(a[, n, axis])	Wrapping of numpy.fft.hfft
fft.ihfft(a[, n, axis])	Wrapping of numpy.fft.ihfft
fft.rfft(a[, n, axis])	Wrapping of numpy.fft.rfft
fft.irfft(a[, n, axis])	Wrapping of numpy.fft.irfft
fix	Round to nearest integer towards zero.
floor(x[, out])	Return the floor of the input, element-wise.
fmax(x1, x2[, out])	Element-wise maximum of array elements.
fmin(x1, x2[, out])	Element-wise minimum of array elements.
fmod(x1, x2[, out])	Return the element-wise remainder of division.
frexp(x)	
fromfunction(func[, chunks, shape, dtype])	Construct an array by executing a function over each coordinate.
full	Blocked variant of full
histogram(a[, bins, range, normed, weights, ...])	
hstack(tup)	Stack arrays in sequence horizontally (column wise).
hypot(x1, x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.
imag	Return the imaginary part of the elements of the array.
insert(arr, obj, values, axis)	Insert values along the given axis before the given indices.
isclose(arr1, arr2[, rtol, atol, equal_nan])	Returns a boolean array where two arrays are element-wise equal within a tolera
iscomplex	Returns a bool array, where True if input element is complex.
isfinite(x[, out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf(x[, out])	Test element-wise for positive or negative infinity.
isnan(x[, out])	Test element-wise for NaN and return result as a boolean array.
isnull(values)	pandas.isnull for dask arrays
isreal	Returns a bool array, where True if input element is real.
ldexp(x1, x2[, out])	Returns x1 * 2**x2, element-wise.
linspace(start, stop[, num, chunks, dtype])	Return *num* evenly spaced values over the closed interval [*start*, *stop*].
log(x[, out])	Natural logarithm, element-wise.
log10(x[, out])	Return the base 10 logarithm of the input array, element-wise.
log1p(x[, out])	Return the natural logarithm of one plus the input array, element-wise.
log2(x[, out])	Base-2 logarithm of *x*.

Table 4.1 – continued from previous page

logaddexp(x1, x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x1, x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
logical_and(x1, x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_not(x[, out])	Compute the truth value of NOT x element-wise.
logical_or(x1, x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_xor(x1, x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
max(a[, axis, keepdims, split_every])	Return the maximum of an array or maximum along an axis.
maximum(x1, x2[, out])	Element-wise maximum of array elements.
mean(a[, axis, dtype, keepdims, split_every])	Compute the arithmetic mean along the specified axis.
min(a[, axis, keepdims, split_every])	Return the minimum of an array or minimum along an axis.
minimum(x1, x2[, out])	Element-wise minimum of array elements.
modf(x)	
moment(a, order[, axis, dtype, keepdims, ...])	
nanargmax(x[, axis, split_every])	
nanargmin(x[, axis, split_every])	
nancumprod(x, axis[, dtype])	Return the cumulative product of array elements over a given axis treating Not a
nancumsum(x, axis[, dtype])	Return the cumulative sum of array elements over a given axis treating Not a Nu
nanmax(a[, axis, keepdims, split_every])	Return the maximum of an array or maximum along an axis, ignoring any NaNs
nanmean(a[, axis, dtype, keepdims, split_every])	Compute the arithmetic mean along the specified axis, ignoring NaNs.
nanmin(a[, axis, keepdims, split_every])	Return minimum of an array or minimum along an axis, ignoring any NaNs.
nanprod(a[, axis, dtype, keepdims, split_every])	Return the product of array elements over a given axis treating Not a Numbers (
nanstd(a[, axis, dtype, keepdims, ddof, ...])	Compute the standard deviation along the specified axis, while ignoring NaNs.
nansum(a[, axis, dtype, keepdims, split_every])	Return the sum of array elements over a given axis treating Not a Numbers (NaI
nanvar(a[, axis, dtype, keepdims, ddof, ...])	Compute the variance along the specified axis, while ignoring NaNs.
nextafter(x1, x2[, out])	Return the next floating-point value after x1 towards x2, element-wise.
notnull(values)	pandas.notnull for dask arrays
ones	Blocked variant of ones
percentile(a, q[, interpolation])	Approximate percentile of 1-D array
prod(a[, axis, dtype, keepdims, split_every])	Return the product of array elements over a given axis.
rad2deg(x[, out])	Convert angles from radians to degrees.
radians(x[, out])	Convert angles from degrees to radians.
random	
ravel(array)	Return a flattened array.
real	Return the real part of the elements of the array.
rechunk(x, chunks)	Convert blocks in dask array x for new chunks.
reshape(array, shape)	Gives a new shape to an array without changing its data.
rint(x[, out])	Round elements of the array to the nearest integer.
sign(x[, out])	Returns an element-wise indication of the sign of a number.
signbit(x[, out])	Returns element-wise True where signbit is set (less than zero).
sin(x[, out])	Trigonometric sine, element-wise.
sinh(x[, out])	Hyperbolic sine, element-wise.
sqrt(x[, out])	Return the positive square-root of an array, element-wise.
square(x[, out])	Return the element-wise square of the input.
squeeze(a[, axis])	Remove single-dimensional entries from the shape of an array.
stack(seq[, axis])	Stack arrays along a new axis
std(a[, axis, dtype, keepdims, ddof, ...])	Compute the standard deviation along the specified axis.
sum(a[, axis, dtype, keepdims, split_every])	Sum of array elements over a given axis.
take(a, indices[, axis])	Take elements from an array along an axis.
tan(x[, out])	Compute tangent element-wise.
tanh(x[, out])	Compute hyperbolic tangent element-wise.

Table 4.1 – continued from previous page

tensordot(lhs, rhs[, axes])	Compute tensor dot product along specified axes for arrays >= 1-D.
topk(k, x)	The top k elements of an array
transpose(a[, axes])	Permute the dimensions of an array.
tril(m[, k])	Lower triangle of an array with elements above the *k*-th diagonal zeroed.
triu(m[, k])	Upper triangle of an array with elements above the *k*-th diagonal zeroed.
trunc(x[, out])	Return the truncated value of the input, element-wise.
unique(x)	Find the unique elements of an array.
var(a[, axis, dtype, keepdims, ddof, ...])	Compute the variance along the specified axis.
vnorm(a[, ord, axis, dtype, keepdims, ...])	Vector norm
vstack(tup)	Stack arrays in sequence vertically (row wise).
where(condition, [x, y])	Return elements, either from *x* or *y*, depending on *condition*.
zeros	Blocked variant of zeros

Linear Algebra

linalg.cholesky(a[, lower])	Returns the Cholesky decomposition, $A = LL^*$ or $A = U^*U$ of a Hermitia
linalg.inv(a)	Compute the inverse of a matrix with LU decomposition and forward / bac
linalg.lstsq(a, b)	Return the least-squares solution to a linear matrix equation using QR deco
linalg.lu(a)	Compute the lu decomposition of a matrix.
linalg.qr(a[, name])	Compute the qr factorization of a matrix.
linalg.solve(a, b[, sym_pos])	Solve the equation `a x = b` for `x`.
linalg.solve_triangular(a, b[, lower])	Solve the equation *a x* = *b* for *x*, assuming a is a triangular matrix.
linalg.svd(a[, name])	Compute the singular value decomposition of a matrix.
linalg.svd_compressed(a, k[, n_power_iter, ...])	Randomly compressed rank-k thin Singular Value Decomposition.
linalg.tsqr(data[, name, compute_svd])	Direct Tall-and-Skinny QR algorithm

Random

random.beta(a, b[, size])	The Beta distribution over `[0, 1]`.
random.binomial(n, p[, size])	Draw samples from a binomial distribution.
random.chisquare(df[, size])	Draw samples from a chi-square distribution.
random.different_seeds(n[, random_state])	A list of different 32 bit integer seeds
random.exponential([scale, size])	Exponential distribution.
random.f(dfnum, dfden[, size])	Draw samples from a F distribution.
random.gamma(shape[, scale, size])	Draw samples from a Gamma distribution.
random.geometric(p[, size])	Draw samples from the geometric distribution.
random.gumbel([loc, scale, size])	Gumbel distribution.
random.hypergeometric(ngood, nbad, nsample)	Draw samples from a Hypergeometric distribution.
random.laplace([loc, scale, size])	Draw samples from the Laplace or double exponential distribution with sp
random.logistic([loc, scale, size])	Draw samples from a Logistic distribution.
random.lognormal([mean, sigma, size])	Return samples drawn from a log-normal distribution.
random.logseries(p[, size])	Draw samples from a Logarithmic Series distribution.
random.negative_binomial(n, p[, size])	Draw samples from a negative_binomial distribution.
random.noncentral_chisquare(df, nonc[, size])	Draw samples from a noncentral chi-square distribution.
random.noncentral_f(dfnum, dfden, nonc[, size])	Draw samples from the noncentral F distribution.
random.normal([loc, scale, size])	Draw random samples from a normal (Gaussian) distribution.
random.pareto(a[, size])	Draw samples from a Pareto II or Lomax distribution with specified shape
random.poisson([lam, size])	Draw samples from a Poisson distribution.

Table 4.3 – continued from previous page

random.power(a[, size])	Draws samples in [0, 1] from a power distribution with positive exponent
random.random([size])	Return random floats in the half-open interval [0.0, 1.0).
random.random_sample([size])	Return random floats in the half-open interval [0.0, 1.0).
random.rayleigh([scale, size])	Draw samples from a Rayleigh distribution.
random.standard_cauchy([size])	Standard Cauchy distribution with mode = 0.
random.standard_exponential([size])	Draw samples from the standard exponential distribution.
random.standard_gamma(shape[, size])	Draw samples from a Standard Gamma distribution.
random.standard_normal([size])	Returns samples from a Standard Normal distribution (mean=0, stdev=1).
random.standard_t(df[, size])	Standard Student's t distribution with df degrees of freedom.
random.triangular(left, mode, right[, size])	Draw samples from the triangular distribution.
random.uniform([low, high, size])	Draw samples from a uniform distribution.
random.vonmises(mu, kappa[, size])	Draw samples from a von Mises distribution.
random.wald(mean, scale[, size])	Draw samples from a Wald, or Inverse Gaussian, distribution.
random.weibull(a[, size])	Weibull distribution.
random.zipf(a[, size])	Standard distributions

Slightly Overlapping Ghost Computations

ghost.ghost(x, depth, boundary)	Share boundaries between neighboring blocks
ghost.map_overlap(x, func, depth[, ...])	

Create and Store Arrays

from_array(x, chunks[, name, lock])	Create dask array from something that looks like an array
from_delayed(value, shape[, dtype, name])	Create a dask array from a dask delayed value
from_npy_stack(dirname[, mmap_mode])	Load dask array from stack of npy files
store(sources, targets[, lock, compute])	Store dask arrays in array-like objects, overwrite data in target
to_hdf5(filename, *args, **kwargs)	Store arrays in HDF5 file
to_npy_stack(dirname, x[, axis])	Write dask array to a stack of .npy files

Internal functions

map_blocks(func, *args, **kwargs)	Map a function across all blocks of a dask array
atop(func, out_ind, *args, **kwargs)	Tensor operation: Generalized inner and outer products
top(func, output, out_indices, ...)	Tensor operation

Other functions

dask.array.**from_array**(*x*, *chunks*, *name=None*, *lock=False*)

Create dask array from something that looks like an array

Input must have a `.shape` and support numpy-style slicing.

The `chunks` argument must be one of the following forms:

- a blocksize like 1000
- a blockshape like (1000, 1000)
- explicit sizes of all blocks along all dimensions like ((1000, 1000, 500), (400, 400)).

Examples

```
>>> x = h5py.File('...')['/data/path']
>>> a = da.from_array(x, chunks=(1000, 1000))
```

If your underlying datastore does not support concurrent reads then include the `lock=True` keyword argument or `lock=mylock` if you want multiple arrays to coordinate around the same lock.

```
>>> a = da.from_array(x, chunks=(1000, 1000), lock=True)
```

dask.array.**from_delayed**(*value*, *shape*, *dtype=None*, *name=None*)
Create a dask array from a dask delayed value

This routine is useful for constructing dask arrays in an ad-hoc fashion using dask delayed, particularly when combined with stack and concatenate.

The dask array will consist of a single chunk.

Examples

```
>>> from dask import do
>>> value = do(np.ones)(5)
>>> array = from_delayed(value, (5,), dtype=float)
>>> array
dask.array<from-va..., shape=(5,), dtype=float64, chunksize=(5,)>
>>> array.compute()
array([ 1.,  1.,  1.,  1.,  1.])
```

dask.array.**store**(*sources*, *targets*, *lock=True*, *compute=True*, ***kwargs*)
Store dask arrays in array-like objects, overwrite data in target

This stores dask arrays into object that supports numpy-style setitem indexing. It stores values chunk by chunk so that it does not have to fill up memory. For best performance you can align the block size of the storage target with the block size of your array.

If your data fits in memory then you may prefer calling `np.array(myarray)` instead.

Parameters **sources: Array or iterable of Arrays**

targets: array-like or iterable of array-likes

These should support setitem syntax `target[10:20] = ...`

lock: boolean or threading.Lock, optional

Whether or not to lock the data stores while storing. Pass True (lock each file individually), False (don't lock) or a particular `threading.Lock` object to be shared among all writes.

compute: boolean, optional

If true compute immediately, return lazy Value object otherwise

Examples

```
>>> x = ...
```

```
>>> import h5py
>>> f = h5py.File('myfile.hdf5')
>>> dset = f.create_dataset('/data', shape=x.shape,
...                                  chunks=x.chunks,
...                                  dtype='f8')
```

```
>>> store(x, dset)
```

Alternatively store many arrays at the same time

```
>>> store([x, y, z], [dset1, dset2, dset3])
```

dask.array.**topk**(*k*, *x*)

The top k elements of an array

Returns the k greatest elements of the array in sorted order. Only works on arrays of a single dimension.

```
>>> x = np.array([5, 1, 3, 6])
>>> d = from_array(x, chunks=2)
>>> d.topk(2).compute()
array([6, 5])
```

Runs in near linear time, returns all results in a single chunk so all k elements must fit in memory.

dask.array.**coarsen**(*reduction*, *x*, *axes*, *trim_excess=False*)

Coarsen array by applying reduction to fixed size neighborhoods

Parameters **reduction: function**

Function like np.sum, np.mean, etc...

x: np.ndarray

Array to be coarsened

axes: dict

Mapping of axis to coarsening factor

Examples

```
>>> x = np.array([1, 2, 3, 4, 5, 6])
>>> coarsen(np.sum, x, {0: 2})
array([ 3,  7, 11])
>>> coarsen(np.max, x, {0: 3})
array([3, 6])
```

Provide dictionary of scale per dimension

```
>>> x = np.arange(24).reshape((4, 6))
>>> x
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

```
>>> coarsen(np.min, x, {0: 2, 1: 3})
array([[ 0,  3],
       [12, 15]])
```

You must avoid excess elements explicitly >>> x = np.array([1, 2, 3, 4, 5, 6, 7, 8]) >>> coarsen(np.min, x, {0: 3}, trim_excess=True) array([1, 4])

dask.array.**stack**(*seq*, *axis=0*)
Stack arrays along a new axis

Given a sequence of dask Arrays form a new dask Array by stacking them along a new dimension (axis=0 by default)

See also:

concatenate

Examples

Create slices

```
>>> import dask.array as da
>>> import numpy as np
```

```
>>> data = [from_array(np.ones((4, 4)), chunks=(2, 2))
...          for i in range(3)]
```

```
>>> x = da.stack(data, axis=0)
>>> x.shape
(3, 4, 4)
```

```
>>> da.stack(data, axis=1).shape
(4, 3, 4)
```

```
>>> da.stack(data, axis=-1).shape
(4, 4, 3)
```

Result is a new dask Array

dask.array.**concatenate**(*seq*, *axis=0*)
Concatenate arrays along an existing axis

Given a sequence of dask Arrays form a new dask Array by stacking them along an existing dimension (axis=0 by default)

See also:

stack

Examples

Create slices

```
>>> import dask.array as da
>>> import numpy as np
```

```
>>> data = [from_array(np.ones((4, 4)), chunks=(2, 2))
...          for i in range(3)]
```

```
>>> x = da.concatenate(data, axis=0)
>>> x.shape
(12, 4)
```

```
>>> da.concatenate(data, axis=1).shape
(4, 12)
```

Result is a new dask Array

dask.array.**all**(*a*, *axis=None*, *keepdims=False*, *split_every=None*)

Test whether all array elements along a given axis evaluate to True.

Parameters **a** : array_like

Input array or object that can be converted to an array.

axis : None or int or tuple of ints, optional

Axis or axes along which a logical AND reduction is performed. The default (*axis* = *None*) is perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

out : ndarray, 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). See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **all** : ndarray, bool

A new boolean or array is returned unless *out* is specified, in which case a reference to *out* is returned.

See also:

ndarray.all equivalent method

any Test whether any element along a given axis evaluates to True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

Examples

```
>>> np.all([[True,False],[True,True]])
False
```

```
>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False], dtype=bool)
```

```
>>> np.all([-1, 4, 5])
True
```

```
>>> np.all([1.0, np.nan])
True
```

```
>>> o=np.array([False])
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array([ True], dtype=bool))
```

dask.array.**angle**(**args*, ***kwargs*)

Return the angle of the complex argument.

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, scalar}

The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

See also:

arctan2, absolute

Examples

```
>>> np.angle([1.0, 1.0j, 1+1j])               # in radians
array([ 0.        ,  1.57079633,  0.78539816])
>>> np.angle(1+1j, deg=True)                  # in degrees
45.0
```

dask.array.**any**(*a*, *axis=None*, *keepdims=False*, *split_every=None*)

Test whether any array element along a given axis evaluates to True.

Returns single boolean unless *axis* is not None

Parameters **a** : array_like

Input array or object that can be converted to an array.

axis : None or int or tuple of ints, optional

Axis or axes along which a logical OR reduction is performed. The default (*axis* = *None*) is perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

out : ndarray, 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 it is of type float, then it will remain so, returning 1.0 for True and 0.0 for False, regardless of the type of *a*). See *doc.ufuncs* (Section "Output arguments") for details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **any** : bool or ndarray

A new boolean or *ndarray* is returned unless *out* is specified, in which case a reference to *out* is returned.

See also:

`ndarray.any` equivalent method

all Test whether all elements along a given axis evaluate to True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

Examples

```
>>> np.any([[True, False], [True, True]])
True
```

```
>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False], dtype=bool)
```

```
>>> np.any([-1, 0, 5])
True
```

```
>>> np.any(np.nan)
True
```

```
>>> o=np.array([False])
>>> z=np.any([-1, 4, 5], out=o)
>>> z, o
(array([ True], dtype=bool), array([ True], dtype=bool))
>>> # Check now that z is a reference to o
>>> z is o
True
>>> id(z), id(o) # identity of z and o
(191614240, 191614240)
```

`dask.array.`**`arange`**(**args*, ***kwargs*)

Return evenly spaced values from *start* to *stop* with step size *step*.

The values are half-open [start, stop), so including start and excluding stop. This is basically the same as python's range function but for dask arrays.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use linspace for these cases.

Parameters **start** : int, optional

The starting value of the sequence. The default is 0.

stop : int

The end of the interval, this value is excluded from the interval.

step : int, optional

The spacing between the values. The default is 1 when not specified. The last value of the sequence.

chunks : int

The number of samples on each block. Note that the last block will have fewer samples if `len(array) % chunks != 0`.

Returns **samples** : dask array

`dask.array.`**`arccos`**(*x*[, *out*])

Trigonometric inverse cosine, element-wise.

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, optional

Array of the same shape as *a*, to store results in. See *doc.ufuncs* (Section "Output arguments") for more details.

Returns **angle** : ndarray

The angle of the ray intersecting the unit circle at the given *x*-coordinate in radians [0, pi]. If *x* is a scalar then a scalar is returned, otherwise an array of the same shape as *x* is returned.

See also:

cos, *arctan*, *arcsin*, `emath.arccos`

Notes

arccos is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cos(z) = x*. The convention is to return the angle *z* whose real part lies in *[0, pi]*.

For real-valued input data types, *arccos* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytic function that has branch cuts *[-inf, -1]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse *cos* is also known as *acos* or cos^-1.

References

M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```

dask.array.**arccosh**(*x*[, *out*])

Inverse hyperbolic cosine, element-wise.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Array of the same shape as *x*, to store results in. See *doc.ufuncs* (Section "Output arguments") for details.

Returns **arccosh** : ndarray

Array of the same shape as *x*.

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, inf]`.

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 *[-inf, 1]* and is continuous from above on it.

References

[R76], *[R77]*

Examples

```
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])
>>> np.arccosh(1)
0.0
```

dask.array.**arcsin**(*x*[, *out*])

Inverse sine, element-wise.

Parameters **x** : array_like

y-coordinate on the unit circle.

out : ndarray, optional

Array of the same shape as *x*, in which to store the results. See *doc.ufuncs* (Section "Output arguments") for more details.

Returns **angle** : ndarray

The inverse sine of each element in *x*, in radians and in the closed interval `[-pi/2, pi/2]`. If *x* is a scalar, a scalar is returned, otherwise an array.

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 [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as *asin* or sin^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. http://www.math.sfu.ca/~cbm/aands/

Examples

```
>>> np.arcsin(1)     # pi/2
1.5707963267948966
>>> np.arcsin(-1)    # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
```

`dask.array.`**`arcsinh`**(*x*[, *out*])

Inverse hyperbolic sine element-wise.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See *doc.ufuncs*.

Returns **out** : ndarray

Array of of the same shape as *x*.

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

[R78], *[R79]*

Examples

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])
```

`dask.array.`**`arctan`**(*x*[, *out*])

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if `y = tan(x)` then `x = arctan(y)`.

Parameters **x** : array_like

Input values. *arctan* is applied to each element of *x*.

Returns **out** : ndarray

Out has the same shape as *x*. Its real part is in `[-pi/2, pi/2]` (`arctan(+/-inf)` returns `+/-pi/2`). It 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. http://www.math.sfu.ca/~cbm/aands/

Examples

We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```
>>> np.arctan([0, 1])
array([ 0.        ,  0.78539816])
```

```
>>> np.pi/4
0.78539816339744828
```

Plot arctan:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
>>> plt.axis('tight')
>>> plt.show()
```

dask.array.**arctan2**(*x1*, *x2*[, *out*])

Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.

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. *x2* must be broadcastable to match the shape of *x1* or vice versa.

Returns **angle** : ndarray

Array of angles in radians, in the range `[-pi, pi]`.

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: *[R80]*

x1	*x2*	*arctan2(x1,x2)*
+/- 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

[R80]

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])
```

`dask.array.`**`arctanh`**(*x*[, *out*])

Inverse hyperbolic tangent element-wise.

Parameters **x** : array_like

Input array.

Returns **out** : ndarray

Array of the same shape as *x*.

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

[R81], *[R82]*

Examples

```
>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

dask.array.**argmax**(*x*, *axis=None*, *split_every=None*)

Indices of the maximum values along an axis.

Parameters **a** : array_like

Input array.

axis : int, optional

By default, the index is into the flattened array, otherwise along the specified axis.

Returns **index_array** : ndarray of ints

Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed.

See also:

ndarray.argmax, *argmin*

amax The maximum value along a given axis.

unravel_index Convert a flat index into an index tuple.

Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])
```

```
>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b) # Only the first occurrence is returned.
1
```

dask.array.**argmin**(*x*, *axis=None*, *split_every=None*)

Return the indices of the minimum values along an axis.

See also:

argmax Similar function. Please refer to *numpy.argmax* for detailed documentation.

dask.array.**around**(*x*, *decimals=0*)

Evenly round to the given number of decimals.

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 *doc.ufuncs* (Section "Output arguments") for 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

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. Results may also be surprising due to the inexact representation of decimal fractions in the IEEE floating point standard *[R83]* and errors introduced when scaling by powers of ten.

References

[R83], *[R84]*

Examples

```
>>> np.around([0.37, 1.64])
array([ 0.,  2.])
>>> np.around([0.37, 1.64], decimals=1)
array([ 0.4,  1.6])
>>> np.around([.5, 1.5, 2.5, 3.5, 4.5]) # rounds to nearest even value
array([ 0.,  2.,  2.,  4.,  4.])
>>> np.around([1,2,3,11], decimals=1) # ndarray of ints is returned
array([ 1,  2,  3, 11])
>>> np.around([1,2,3,11], decimals=-1)
array([ 0,  0,  0, 10])
```

dask.array.**array**(*object*, *dtype=None*, *copy=True*, *order=None*, *subok=False*, *ndmin=0*)

Create an array.

Parameters **object** : array_like

An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype : data-type, optional

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

copy : bool, optional

If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*dtype*, *order*, etc.).

order : {'C', 'F', 'A'}, optional

Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

subok : bool, optional

If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

ndmin : int, optional

Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

Returns **out** : ndarray

An array object satisfying the specified requirements.

See also:

empty, empty_like, *zeros*, zeros_like, *ones*, ones_like, fill

Examples

```
>>> np.array([1, 2, 3])
array([1, 2, 3])
```

Upcasting:

```
>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])
```

More than one dimension:

```
>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```

Minimum dimensions 2:

```
>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])
```

Type provided:

```
>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Data-type consisting of more than one element:

```
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])
```

Creating an array from sub-classes:

```
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])
```

```
>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
```

`dask.array.`**`bincount`**(*x*, *weights=None*, *minlength=None*)

Count number of occurrences of each value in array of non-negative ints.

The number of bins (of size 1) is one larger than the largest value in *x*. If *minlength* is specified, there will be at least this number of bins in the output array (though it will be longer if necessary, depending on the contents of *x*). Each bin gives the number of occurrences of its index value in *x*. If *weights* is specified the input array is weighted by it, i.e. if a value `n` is found at position `i`, `out[n] += weight[i]` instead of `out[n] += 1`.

Parameters **x** : array_like, 1 dimension, nonnegative ints

Input array.

weights : array_like, optional

Weights, array of the same shape as *x*.

minlength : int, optional

New in version 1.6.0.

A minimum number of bins for the output array.

Returns **out** : ndarray of ints

The result of binning the input array. The length of *out* is equal to `np.amax(x)+1`.

Raises **ValueError**

If the input is not 1-dimensional, or contains elements with negative values, or if *minlength* is non-positive.

TypeError

If the type of the input is float or complex.

See also:

histogram, `digitize`, *unique*

Examples

```
>>> np.bincount(np.arange(5))
array([1, 1, 1, 1, 1])
>>> np.bincount(np.array([0, 1, 1, 3, 2, 1, 7]))
array([1, 3, 1, 1, 0, 0, 0, 1])
```

```
>>> x = np.array([0, 1, 1, 3, 2, 1, 7, 23])
>>> np.bincount(x).size == np.amax(x)+1
True
```

The input array needs to be of integer dtype, otherwise a TypeError is raised:

```
>>> np.bincount(np.arange(5, dtype=np.float))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: array cannot be safely cast to required type
```

A possible use of `bincount` is to perform sums over variable-size chunks of an array, using the `weights` keyword.

```
>>> w = np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6]) # weights
>>> x = np.array([0, 1, 1, 2, 2, 2])
>>> np.bincount(x,  weights=w)
array([ 0.3,  0.7,  1.1])
```

`dask.array.`**`broadcast_to`**(*x*, *shape*)

Broadcast an array to a new shape.

Parameters **array** : array_like

The array to broadcast.

shape : tuple

The shape of the desired array.

subok : bool, optional

If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

Returns **broadcast** : array

A readonly view on the original array with the given shape. It is typically not contiguous. Furthermore, more than one element of a broadcasted array may refer to a single memory location.

Raises **ValueError**

If the array is not compatible with the new shape according to NumPy's broadcasting rules.

Examples

```
>>> x = np.array([1, 2, 3])
>>> np.broadcast_to(x, (3, 3))
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
```

dask.array.**coarsen**(*reduction*, *x*, *axes*, *trim_excess=False*)

Coarsen array by applying reduction to fixed size neighborhoods

Parameters **reduction: function**

Function like np.sum, np.mean, etc...

x: np.ndarray

Array to be coarsened

axes: dict

Mapping of axis to coarsening factor

Examples

```
>>> x = np.array([1, 2, 3, 4, 5, 6])
>>> coarsen(np.sum, x, {0: 2})
array([ 3,  7, 11])
>>> coarsen(np.max, x, {0: 3})
array([3, 6])
```

Provide dictionary of scale per dimension

```
>>> x = np.arange(24).reshape((4, 6))
>>> x
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

```
>>> coarsen(np.min, x, {0: 2, 1: 3})
array([[ 0,  3],
       [12, 15]])
```

You must avoid excess elements explicitly >>> x = np.array([1, 2, 3, 4, 5, 6, 7, 8]) >>> coarsen(np.min, x, {0: 3}, trim_excess=True) array([1, 4])

dask.array.**ceil**(*x*[, *out*])

Return the ceiling of the input, element-wise.

The ceil of the scalar *x* is the smallest integer *i*, such that *i* >= *x*. It is often denoted as $\lceil x \rceil$.

Parameters **x** : array_like

Input data.

Returns **y** : {ndarray, scalar}

The ceiling of each element in *x*, with *float* dtype.

See also:

floor, *trunc*, *rint*

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.ceil(a)
array([-1., -1., -0.,  1.,  2.,  2.,  2.])
```

dask.array.**choose**(*a*, *choices*)

Construct an array from an index array and a set of arrays to choose from.

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = *numpy.lib.index_tricks*):

`np.choose(a,c) == np.array([c[a[I]][I] for I in ndi.ndindex(a.shape)]).`

But this omits some subtleties. Here is a fully general summary:

Given an "index" array (*a*) of integers and a sequence of *n* arrays (*choices*), *a* and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these *Ba* and *Bchoices[i], i = 0,...,n-1* we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each *i*. Then, a new array with shape `Ba.shape` is created as follows:

- if `mode=raise` (the default), then, first of all, each element of *a* (and thus *Ba*) must be in the range *[0, n-1]*; now, suppose that *i* (in that range) is the value at the *(j0, j1, ..., jm)* position in *Ba* - then the value at the same position in the new array is the value in *Bchoices[i]* at that same position;
- if `mode=wrap`, values in *a* (and thus *Ba*) may be any (signed) integer; modular arithmetic is used to map integers outside the range *[0, n-1]* back into that range; and then the new array is constructed as above;
- if `mode=clip`, values in *a* (and thus *Ba*) may be any (signed) integer; negative integers are mapped to 0; values greater than *n-1* are mapped to *n-1*; and then the new array is constructed as above.

Parameters **a** : int array

This array must contain integers in *[0, n-1]*, where *n* is the number of choices, unless `mode=wrap` or `mode=clip`, in which cases any integers are permissible.

choices : sequence of arrays

Choice arrays. *a* and all of the choices must be broadcastable to the same shape. If *choices* is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the "sequence".

out : array, optional

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

mode : {'raise' (default), 'wrap', 'clip'}, optional

Specifies how indices outside *[0, n-1]* will be treated:

- 'raise' : an exception is raised
- 'wrap' : value becomes value mod *n*
- 'clip' : values < 0 are mapped to 0, values > n-1 are mapped to n-1

Returns **merged_array** : array

The merged result.

Raises **ValueError: shape mismatch**

If *a* and each choice array are not all broadcastable to the same shape.

See also:

ndarray.choose equivalent method

Notes

To reduce the chance of misinterpretation, even though the following "abuse" is nominally supported, *choices* should neither be, nor be thought of as, a single array, i.e., the outermost sequence-like container should be either a list or a tuple.

Examples

```
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...   [20, 21, 22, 23], [30, 31, 32, 33]]
>>> np.choose([2, 3, 1, 0], choices
... # the first element of the result will be the first element of the
... # third (2+1) "array" in choices, namely, 20; the second element
... # will be the second element of the fourth (3+1) choice array, i.e.,
... # 31, etc.
... )
array([20, 31, 12,  3])
>>> np.choose([2, 4, 1, 0], choices, mode='clip') # 4 goes to 3 (4-1)
array([20, 31, 12,  3])
>>> # because there are 4 choice arrays
>>> np.choose([2, 4, 1, 0], choices, mode='wrap') # 4 goes to (4 mod 4)
array([20,  1, 12,  3])
>>> # i.e., 0
```

A couple examples illustrating how choose broadcasts:

```
>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
>>> choices = [-10, 10]
>>> np.choose(a, choices)
array([[ 10, -10,  10],
       [-10,  10, -10],
       [ 10, -10,  10]])
```

```
>>> # With thanks to Anne Archibald
>>> a = np.array([0, 1]).reshape((2,1,1))
>>> c1 = np.array([1, 2, 3]).reshape((1,3,1))
>>> c2 = np.array([-1, -2, -3, -4, -5]).reshape((1,1,5))
>>> np.choose(a, (c1, c2)) # result is 2x3x5, res[0,:,:]=c1, res[1,:,:]=c2
array([[[ 1,  1,  1,  1,  1],
        [ 2,  2,  2,  2,  2],
        [ 3,  3,  3,  3,  3]],
       [[-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5]]])
```

`dask.array.`**`clip`**(**args*, ***kwargs*)

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters **a** : array_like

Array containing elements to clip.

a_min : scalar or array_like

Minimum value.

a_max : scalar or array_like

Maximum value. If *a_min* or *a_max* are array_like, then they will be broadcasted to the shape of *a*.

out : ndarray, optional

The results will be placed in this array. It may be the input array for in-place clipping. *out* must be of the right shape to hold the output. Its type is preserved.

Returns **clipped_array** : ndarray

An array with the elements of *a*, but where values < *a_min* are replaced with *a_min*, and those > *a_max* with *a_max*.

See also:

numpy.doc.ufuncs Section "Output arguments"

Examples

```
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3,4,1,1,1,4,4,4,4,4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

dask.array.**compress**(*condition*, *a*, *axis=None*)

Return selected slices of an array along given axis.

When working along a given axis, a slice along that axis is returned in *output* for each index where *condition* evaluates to True. When working on a 1-D array, *compress* is equivalent to *extract*.

Parameters **condition** : 1-D array of bools

Array that selects which entries to return. If len(condition) is less than the size of *a* along the given axis, then output is truncated to the length of the condition array.

a : array_like

Array from which to extract a part.

axis : int, optional

Axis along which to take slices. If None (default), work on the flattened array.

out : ndarray, optional

Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns **compressed_array** : ndarray

A copy of *a* without the slices along axis for which *condition* is false.

See also:

take, *choose*, *diag*, diagonal, select

ndarray.compress Equivalent method in ndarray

np.extract Equivalent method when working on 1-D arrays

numpy.doc.ufuncs Section "Output arguments"

Examples

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.compress([0, 1], a, axis=0)
array([[3, 4]])
>>> np.compress([False, True, True], a, axis=0)
array([[3, 4],
       [5, 6]])
>>> np.compress([False, True], a, axis=1)
array([[2],
       [4],
       [6]])
```

Working on the flattened array does not return slices along an axis but selects elements.

```
>>> np.compress([False, True], a)
array([2])
```

dask.array.**concatenate**(*seq*, *axis=0*)

Concatenate arrays along an existing axis

Given a sequence of dask Arrays form a new dask Array by stacking them along an existing dimension (axis=0 by default)

See also:

stack

Examples

Create slices

```
>>> import dask.array as da
>>> import numpy as np
```

```
>>> data = [from_array(np.ones((4, 4)), chunks=(2, 2))
...          for i in range(3)]
```

```
>>> x = da.concatenate(data, axis=0)
>>> x.shape
(12, 4)
```

```
>>> da.concatenate(data, axis=1).shape
(4, 12)
```

Result is a new dask Array

dask.array.**conj**(*x*[, *out*])

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Parameters **x** : array_like

Input value.

Returns **y** : ndarray

The complex conjugate of *x*, with same dtype as *y*.

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]])
```

dask.array.**copysign**(*x1*, *x2*[, *out*])

Change the sign of x1 to that of x2, element-wise.

If both arguments are arrays or sequences, they have to be of the same length. 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*.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns **out** : array_like

The values of *x1* with the sign of *x2*.

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.])
```

dask.array.**corrcoef**(*x*, *y=None*, *rowvar=1*)

Return correlation coefficients.

Please refer to the documentation for *cov* for more detail. The relationship between the correlation coefficient matrix, *P*, and the covariance matrix, *C*, is

$$P_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

The values of *P* are between -1 and 1, inclusive.

Parameters **x** : array_like

A 1-D or 2-D array containing multiple variables and observations. Each row of *m* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

y : array_like, optional

An additional set of variables and observations. *y* has the same shape as *m*.

rowvar : int, optional

If *rowvar* is non-zero (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

bias : int, optional

Default normalization is by `(N - 1)`, where `N` is the number of observations (unbiased estimate). If *bias* is 1, then normalization is by `N`. These values can be overridden by using the keyword `ddof` in numpy versions >= 1.5.

ddof : {None, int}, optional

New in version 1.5.

If not `None` normalization is by `(N - ddof)`, where `N` is the number of observations; this overrides the value implied by `bias`. The default value is `None`.

Returns **out** : ndarray

The correlation coefficient matrix of the variables.

See also:

cov Covariance matrix

dask.array.**cos**(*x*[, *out*])

Cosine element-wise.

Parameters **x** : array_like

Input array in radians.

out : ndarray, optional

Output array of same shape as *x*.

Returns **y** : ndarray

The corresponding cosine values.

Raises **ValueError: invalid return array shape**

if *out* is provided and *out.shape* != *x.shape* (See Examples)

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
>>> 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: invalid return array shape
```

`dask.array.`**`cosh`**(*x*[, *out*])

Hyperbolic cosine, element-wise.

Equivalent to `1/2 * (np.exp(x) + np.exp(-x))` and `np.cos(1j*x)`.

Parameters **x** : array_like

Input array.

Returns **out** : ndarray

Output array of same shape as *x*.

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()
```

`dask.array.`**`cov`**(*m*, *y=None*, *rowvar=1*, *bias=0*, *ddof=None*)

Estimate a covariance matrix, given data.

Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, $X = [x_1, x_2, ...x_N]^T$, then the covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i.

Parameters **m** : array_like

A 1-D or 2-D array containing multiple variables and observations. Each row of *m* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

y : array_like, optional

An additional set of variables and observations. *y* has the same form as that of *m*.

rowvar : int, optional

If *rowvar* is non-zero (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

bias : int, optional

Default normalization is by `(N - 1)`, where `N` is the number of observations given (unbiased estimate). If *bias* is 1, then normalization is by `N`. These values can be overridden by using the keyword `ddof` in numpy versions >= 1.5.

ddof : int, optional

New in version 1.5.

If not `None` normalization is by `(N - ddof)`, where `N` is the number of observations; this overrides the value implied by `bias`. The default value is `None`.

Returns **out** : ndarray

The covariance matrix of the variables.

See also:

corrcoef Normalized covariance matrix

Examples

Consider two variables, x_0 and x_1, which correlate perfectly, but in opposite directions:

```
>>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
>>> x
array([[0, 1, 2],
       [2, 1, 0]])
```

Note how x_0 increases while x_1 decreases. The covariance matrix shows this clearly:

```
>>> np.cov(x)
array([[ 1., -1.],
       [-1.,  1.]])
```

Note that element $C_{0,1}$, which shows the correlation between x_0 and x_1, is negative.

Further, note how *x* and *y* are combined:

```
>>> x = [-2.1, -1,  4.3]
>>> y = [3,  1.1,  0.12]
>>> X = np.vstack((x,y))
>>> print np.cov(X)
[[ 11.71        -4.286     ]
 [ -4.286        2.14413333]]
>>> print np.cov(x, y)
[[ 11.71        -4.286     ]
```

```
 [ -4.286        2.14413333]]
>>> print np.cov(x)
11.71
```

dask.array.**cumprod**(*x*, *axis*, *dtype=None*)

Return the cumulative product of elements along a given axis.

Parameters **a** : array_like

Input array.

axis : int, optional

Axis along which the cumulative product is computed. By default the input is flattened.

dtype : dtype, optional

Type of the returned array, as well as of the accumulator in which the elements are multiplied. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

Returns **cumprod** : ndarray

A new array holding the result is returned unless *out* is specified, in which case a reference to out is returned.

See also:

numpy.doc.ufuncs Section "Output arguments"

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> a = np.array([1,2,3])
>>> np.cumprod(a) # intermediate results 1, 1*2
...               # total product 1*2*3 = 6
array([1, 2, 6])
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.cumprod(a, dtype=float) # specify type of output
array([   1.,    2.,    6.,   24.,  120.,  720.])
```

The cumulative product for each column (i.e., over the rows) of *a*:

```
>>> np.cumprod(a, axis=0)
array([[ 1,  2,  3],
       [ 4, 10, 18]])
```

The cumulative product for each row (i.e. over the columns) of *a*:

```
    >>> np.cumprod(a,axis=1)
    array([[  1,   2,   6],
           [  4,  20, 120]])
```

dask.array.**cumsum**(*x*, *axis*, *dtype=None*)

Return the cumulative sum of the elements along a given axis.

Parameters **a** : array_like

Input array.

axis : int, optional

Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.

dtype : dtype, optional

Type of the returned array and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See *doc.ufuncs* (Section "Output arguments") for more details.

Returns **cumsum_along_axis** : ndarray.

A new array holding the result is returned unless *out* is specified, in which case a reference to *out* is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

See also:

sum Sum array elements.

trapz Integration of array values using the composite trapezoidal rule.

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

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
    >>> a = np.array([[1,2,3], [4,5,6]])
    >>> a
    array([[1, 2, 3],
           [4, 5, 6]])
    >>> np.cumsum(a)
    array([ 1,  3,  6, 10, 15, 21])
    >>> np.cumsum(a, dtype=float)     # specifies type of output value(s)
    array([  1.,   3.,   6.,  10.,  15.,  21.])
```

```
>>> np.cumsum(a,axis=0)       # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> np.cumsum(a,axis=1)       # sum over columns for each of the 2 rows
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

dask.array.**deg2rad**(*x*[, *out*])

Convert angles from degrees to radians.

Parameters **x** : array_like

Angles in degrees.

Returns **y** : ndarray

The corresponding angle in radians.

See also:

rad2deg Convert angles from radians to degrees.

unwrap Remove large jumps in angle by wrapping.

Notes

New in version 1.3.0.

`deg2rad(x)` is `x * pi / 180`.

Examples

```
>>> np.deg2rad(180)
3.1415926535897931
```

dask.array.**degrees**(*x*[, *out*])

Convert angles from radians to degrees.

Parameters **x** : array_like

Input array in radians.

out : ndarray, optional

Output array of same shape as x.

Returns **y** : ndarray of floats

The corresponding degree values; if *out* was supplied this is a reference to it.

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 = degrees(rad, out)
>>> np.all(r == out)
True
```

dask.array.**diag**(*v*)

Extract a diagonal or construct a diagonal array.

See the more detailed documentation for `numpy.diagonal` if you use this function to extract a diagonal and wish to write to the resulting array; whether it returns a copy or a view depends on what version of numpy you are using.

Parameters **v** : array_like

If *v* is a 2-D array, return a copy of its *k*-th diagonal. If *v* is a 1-D array, return a 2-D array with *v* on the *k*-th diagonal.

k : int, optional

Diagonal in question. The default is 0. Use *k>0* for diagonals above the main diagonal, and *k<0* for diagonals below the main diagonal.

Returns **out** : ndarray

The extracted diagonal or constructed diagonal array.

See also:

diagonal Return specified diagonals.

diagflat Create a 2-D array with the flattened input as a diagonal.

trace Sum along diagonals.

triu Upper triangle of an array.

tril Lower triange of an array.

Examples

```
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
>>> np.diag(x)
array([0, 4, 8])
>>> np.diag(x, k=1)
array([1, 5])
>>> np.diag(x, k=-1)
array([3, 7])
```

```
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

dask.array.**dot**(*a*, *b*, *out=None*)

Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of *a* and the second-to-last of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters **a** : array_like

First argument.

b : array_like

Second argument.

out : ndarray, optional

Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns **output** : ndarray

Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned.

Raises **ValueError**

If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

See also:

vdot Complex-conjugating dot product.

tensordot Sum products over arbitrary axes.

einsum Einstein summation convention.

Examples

```
>>> np.dot(3, 4)
12
```

Neither argument is complex-conjugated:

```
>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

For 2-D arrays it's the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

dask.array.**dstack**(*tup*)

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by *dsplit*. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

Parameters **tup** : sequence of arrays

Arrays to stack. All of them must have the same shape along all but the third axis.

Returns **stacked** : ndarray

The array formed by stacking the given arrays.

See also:

vstack Stack along first axis.

hstack Stack along second axis.

concatenate Join arrays.

dsplit Split array along third axis.

Notes

Equivalent to `np.concatenate(tup, axis=2)`.

Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
```

```
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
       [[2, 3]],
       [[3, 4]]])
```

dask.array.**empty**()

Blocked variant of empty

Follows the signature of empty exactly except that it also requires a keyword argument chunks=(...)

Original signature follows below. empty(shape, dtype=float, order='C')

Return a new array of given shape and type, without initializing entries.

Parameters **shape** : int or tuple of int

Shape of the empty array

dtype : data-type, optional

Desired output data-type.

order : {'C', 'F'}, optional

Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

See also:

empty_like, *zeros*, *ones*

Notes

empty, unlike *zeros*, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples

```
>>> np.empty([2, 2])
array([[ -9.74499359e+001,   6.69583040e-309],
       [  2.13182611e-314,   3.06959433e-309]])         #random
```

```
>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],
       [  496041986,    19249760]])                     #random
```

dask.array.**exp**(*x*[, *out*])

Calculate the exponential of all elements in the input array.

Parameters **x** : array_like

Input values.

Returns **out** : ndarray

Output array, element-wise exponential of *x*.

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

[R85], *[R86]*

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])
>>> 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])
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```

`dask.array.`**`expm1`**(*x*[, *out*])

Calculate `exp(x) - 1` for all elements in the array.

Parameters **x** : array_like

Input values.

Returns **out** : ndarray

Element-wise exponential minus one: `out = exp(x) - 1`.

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 `1.00000000005e-10` to about 32 significant digits. This example shows the superiority of expm1 in this case.

```
>>> np.expm1(1e-10)
1.00000000005e-10
>>> np.exp(1e-10) - 1
1.000000082740371e-10
```

`dask.array.`**`eye`**(*N*, *chunks*, *M=None*, *k=0*, *dtype=<type 'float'>*)

Return a 2-D Array with ones on the diagonal and zeros elsewhere.

Parameters **N** : int

Number of rows in the output.

chunks: int

chunk size of resulting blocks

M : int, optional

Number of columns in the output. If None, defaults to *N*.

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.

dtype : data-type, optional

Data-type of the returned array.

Returns **I** : Array of shape (N,M)

An array where all elements are equal to zero, except for the *k*-th diagonal, whose values are equal to one.

`dask.array.`**`fabs`**(*x*[, *out*])

Compute the absolute values element-wise.

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, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns **y** : {ndarray, scalar}

The absolute values of *x*, the returned values are always floats.

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])
```

dask.array.**fix**(**args*, ***kwargs*)

Round to nearest integer towards zero.

Round an array of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

Parameters **x** : array_like

An array of floats to be rounded

y : ndarray, optional

Output array

Returns **out** : ndarray of floats

The array of rounded numbers

See also:

trunc, *floor*, *ceil*

around Round to given number of decimals

Examples

```
>>> np.fix(3.14)
3.0
>>> np.fix(3)
3.0
>>> np.fix([2.1, 2.9, -2.1, -2.9])
array([ 2.,  2., -2., -2.])
```

dask.array.**floor**(*x*[, *out*])

Return the floor of the input, element-wise.

The floor of the scalar *x* is the largest integer *i*, such that *i* <= *x*. It is often denoted as $\lfloor x \rfloor$.

Parameters **x** : array_like

Input data.

Returns **y** : {ndarray, scalar}

The floor of each element in *x*.

See also:

ceil, *trunc*, *rint*

Notes

Some spreadsheet programs calculate the "floor-towards-zero", in other words `floor(-2.5) == -2`. NumPy instead uses the definition of *floor* where *floor(-2.5) == -3*.

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.floor(a)
array([-2., -2., -1.,  0.,  1.,  1.,  2.])
```

dask.array.**fmax**(*x1*, *x2*[, *out*])

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 the non-nan 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 ignored when possible.

Parameters **x1, x2** : array_like

The arrays holding the elements to be compared. They must have the same shape.

Returns **y** : {ndarray, scalar}

The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

fmin Element-wise minimum of two arrays, ignores NaNs.

maximum Element-wise maximum of two arrays, propagates NaNs.

amax The maximum value of an array along a given axis, propagates NaNs.

nanmax The maximum value of an array along a given axis, ignores NaNs.

minimum, amin, *nanmin*

Notes

New in version 1.3.0.

The fmax is equivalent to `np.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

Examples

```
>>> np.fmax([2, 3, 4], [1, 5, 2])
array([ 2.,  5.,  4.])
```

```
>>> np.fmax(np.eye(2), [0.5, 2])
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.fmax([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([  0.,   0.,  NaN])
```

dask.array.**fmin**(*x1*, *x2*[, *out*])

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 the non-nan 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 ignored when possible.

Parameters **x1, x2** : array_like

The arrays holding the elements to be compared. They must have the same shape.

Returns **y** : {ndarray, scalar}

The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

fmax Element-wise maximum of two arrays, ignores NaNs.

minimum Element-wise minimum of two arrays, propagates NaNs.

amin The minimum value of an array along a given axis, propagates NaNs.

nanmin The minimum value of an array along a given axis, ignores NaNs.

maximum, amax, *nanmax*

Notes

New in version 1.3.0.

The fmin is equivalent to `np.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

Examples

```
>>> np.fmin([2, 3, 4], [1, 5, 2])
array([2, 5, 4])
```

```
>>> np.fmin(np.eye(2), [0.5, 2])
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.fmin([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([  0.,   0.,  NaN])
```

`dask.array.`**fmod**(*x1*, *x2*[, *out*])

Return the element-wise remainder of division.

This is the NumPy implementation of the C library function fmod, the remainder has the same sign as the dividend *x1*. It is equivalent to the Matlab(TM) `rem` function and should not be confused with the Python modulus operator `x1 % x2`.

Parameters **x1** : array_like

Dividend.

x2 : array_like

Divisor.

Returns **y** : array_like

The remainder of the division of *x1* by *x2*.

See also:

remainder Equivalent to the Python `%` operator.

`divide`

Notes

The result of the modulo operation for negative dividend and divisors is bound by conventions. For *fmod*, the sign of result is the sign of the dividend, while for *remainder* the sign of the result is the sign of the divisor. The *fmod* function is equivalent to the Matlab(TM) `rem` function.

Examples

```
>>> np.fmod([-3, -2, -1, 1, 2, 3], 2)
array([-1,  0, -1,  1,  0,  1])
>>> np.remainder([-3, -2, -1, 1, 2, 3], 2)
array([1, 0, 1, 1, 0, 1])
```

```
>>> np.fmod([5, 3], [2, 2.])
array([ 1.,  1.])
>>> a = np.arange(-3, 3).reshape(3, 2)
>>> a
array([[-3, -2],
       [-1,  0],
       [ 1,  2]])
>>> np.fmod(a, [2,2])
array([[-1,  0],
       [-1,  0],
       [ 1,  0]])
```

`dask.array.`**frexp**(*x*)

`dask.array.`**fromfunction**(*func*, *chunks=None*, *shape=None*, *dtype=None*)

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value `fn(x, y, z)` at coordinate `(x, y, z)`.

Parameters **function** : callable

The function is called with N parameters, where N is the rank of *shape*. Each parameter represents the coordinates of the array varying along a specific axis. For example, if *shape* were `(2, 2)`, then the parameters in turn be (0, 0), (0, 1), (1, 0), (1, 1).

shape : (N,) tuple of ints

Shape of the output array, which also determines the shape of the coordinate arrays passed to *function*.

dtype : data-type, optional

Data-type of the coordinate arrays passed to *function*. By default, *dtype* is float.

Returns **fromfunction** : any

The result of the call to *function* is passed back directly. Therefore the shape of *fromfunction* is completely determined by *function*. If *function* returns a scalar value, the shape of *fromfunction* would match the *shape* parameter.

See also:

`indices`, `meshgrid`

Notes

Keywords other than *dtype* are passed to *function*.

Examples

```
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)
```

```
>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

`dask.array.`**`full`**(**args*, ***kwargs*)

Blocked variant of full

Follows the signature of full exactly except that it also requires a keyword argument chunks=(...)

Original signature follows below.

Return a new array of given shape and type, filled with *fill_value*.

Parameters **shape** : int or sequence of ints

Shape of the new array, e.g., `(2, 3)` or `2`.

fill_value : scalar

Fill value.

dtype : data-type, optional

The desired data-type for the array, e.g., *numpy.int8*. Default is is chosen as *np.array(fill_value).dtype*.

order : {'C', 'F'}, optional

Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns **out** : ndarray

Array of *fill_value* with the given shape, dtype, and order.

See also:

`zeros_like` Return an array of zeros with shape and type of input.

`ones_like` Return an array of ones with shape and type of input.

`empty_like` Return an empty array with shape and type of input.

`full_like` Fill an array with shape and type of input.

`zeros` Return a new array setting values to zero.

ones Return a new array setting values to one.

empty Return a new uninitialized array.

Examples

```
>>> np.full((2, 2), np.inf)
array([[ inf,  inf],
       [ inf,  inf]])
>>> np.full((2, 2), 10, dtype=np.int)
array([[10, 10],
       [10, 10]])
```

dask.array.**histogram**(*a*, *bins=None*, *range=None*, *normed=False*, *weights=None*, *density=None*)

dask.array.**hstack**(*tup*)

Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by *hsplit*.

Parameters **tup** : sequence of ndarrays

All arrays must have the same shape along all but the second axis.

Returns **stacked** : ndarray

The array formed by stacking the given arrays.

See also:

vstack Stack arrays in sequence vertically (row wise).

dstack Stack arrays in sequence depth wise (along third axis).

concatenate Join a sequence of arrays together.

hsplit Split array along second axis.

Notes

Equivalent to `np.concatenate(tup, axis=1)`

Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.hstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

dask.array.**hypot**(*x1*, *x2*[, *out*])

Given the "legs" of a right triangle, return its hypotenuse.

Equivalent to `sqrt(x1**2 + x2**2)`, element-wise. If *x1* or *x2* is scalar_like (i.e., unambiguously castable to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

Parameters **x1, x2** : array_like

Leg of the triangle(s).

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns **z** : ndarray

The hypotenuse of the triangle(s).

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.]])
```

`dask.array.`**`imag`**(**args*, ***kwargs*)

Return the imaginary part of the elements of the array.

Parameters **val** : array_like

Input array.

Returns **out** : ndarray

Output array. 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])
```

`dask.array.`**`insert`**(*arr*, *obj*, *values*, *axis*)

Insert values along the given axis before the given indices.

Parameters **arr** : array_like

Input array.

obj : int, slice or sequence of ints

Object that defines the index or indices before which *values* is inserted.

New in version 1.8.0.

Support for multiple insertions when *obj* is a single scalar or a sequence with one element (similar to calling insert multiple times).

values : array_like

Values to insert into *arr*. If the type of *values* is different from that of *arr*, *values* is converted to the type of *arr*. *values* should be shaped so that `arr[...,obj,...] = values` is legal.

axis : int, optional

Axis along which to insert *values*. If *axis* is None then *arr* is flattened first.

Returns **out** : ndarray

A copy of *arr* with *values* inserted. Note that *insert* does not occur in-place: a new array is returned. If *axis* is None, *out* is a flattened array.

See also:

append Append elements at the end of an array.

concatenate Join a sequence of arrays together.

delete Delete elements from an array.

Notes

Note that for higher dimensional inserts *obj=0* behaves very different from *obj=[0]* just like *arr[:,0,:] = values* is different from *arr[:,[0],:] = values*.

Examples

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([1, 5, 1, 2, 2, 3, 3])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
```

Difference between sequence and scalars: >>> np.insert(a, [1], [[1],[2],[3]], axis=1) array([[1, 1, 1],

[2, 2, 2], [3, 3, 3]])

```
>>> np.array_equal(np.insert(a, 1, [1, 2, 3], axis=1),
...                np.insert(a, [1], [[1],[2],[3]], axis=1))
True
```

```
>>> b = a.flatten()
>>> b
array([1, 1, 2, 2, 3, 3])
>>> np.insert(b, [2, 2], [5, 6])
array([1, 1, 5, 6, 2, 2, 3, 3])
```

```
>>> np.insert(b, slice(2, 4), [5, 6])
array([1, 1, 5, 2, 6, 2, 3, 3])
```

```
>>> np.insert(b, [2, 2], [7.13, False]) # type casting
array([1, 1, 7, 0, 2, 2, 3, 3])
```

```
>>> x = np.arange(8).reshape(2, 4)
>>> idx = (1, 3)
>>> np.insert(x, idx, 999, axis=1)
array([[  0, 999,   1,   2, 999,   3],
       [  4, 999,   5,   6, 999,   7]])
```

dask.array.**isclose**(*arr1*, *arr2*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns a boolean array where two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference (*rtol* * abs(*b*)) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

Parameters **a, b** : array_like

Input arrays to compare.

rtol : float

The relative tolerance parameter (see Notes).

atol : float

The absolute tolerance parameter (see Notes).

equal_nan : bool

Whether to compare NaN's as equal. If True, NaN's in *a* will be considered equal to NaN's in *b* in the output array.

Returns **y** : array_like

Returns a boolean array of where *a* and *b* are equal within the given tolerance. If both *a* and *b* are scalars, returns a single boolean value.

See also:

allclose

Notes

New in version 1.7.0.

For finite values, isclose uses the following equation to test whether two floating point values are equivalent.

absolute(*a* - *b*) <= (*atol* + *rtol* * absolute(*b*))

The above equation is not symmetric in *a* and *b*, so that *isclose(a, b)* might be different from *isclose(b, a)* in some rare cases.

Examples

```
>>> np.isclose([1e10,1e-7], [1.00001e10,1e-8])
array([True, False])
>>> np.isclose([1e10,1e-8], [1.00001e10,1e-9])
array([True, True])
>>> np.isclose([1e10,1e-8], [1.0001e10,1e-9])
array([False, True])
>>> np.isclose([1.0, np.nan], [1.0, np.nan])
array([True, False])
>>> np.isclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
array([True, True])
```

dask.array.**iscomplex**()

Returns a bool array, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

Parameters **x** : array_like

Input array.

Returns **out** : ndarray of bools

Output array.

See also:

isreal

iscomplexobj Return True if x is a complex type or an array of complex numbers.

Examples

```
>>> np.iscomplex([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([ True, False, False, False, False,  True], dtype=bool)
```

dask.array.**isfinite**(*x*[, *out*])

Test element-wise for finiteness (not infinity or not Not a Number).

The result is returned as a boolean array.

Parameters **x** : array_like

Input values.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See *doc.ufuncs*.

Returns **y** : ndarray, bool

For scalar input, the result is a new boolean with value True if the input is finite; otherwise the value is False (input is either positive infinity, negative infinity or Not a Number).

For array input, the result is a boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is finite; otherwise the values are False (element is either positive infinity, negative infinity or Not a Number).

See also:

isinf, isneginf, isposinf, *isnan*

Notes

Not a Number, positive infinity and negative infinity are considered to be non-finite.

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity. Errors result if the second argument is also supplied when *x* is a scalar input, or if first and second arguments have different shapes.

Examples

```
>>> np.isfinite(1)
True
>>> np.isfinite(0)
True
>>> np.isfinite(np.nan)
False
>>> np.isfinite(np.inf)
False
>>> np.isfinite(np.NINF)
False
>>> np.isfinite([np.log(-1.),1.,np.log(0)])
array([False,  True, False], dtype=bool)
```

```
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isfinite(x, y)
array([0, 1, 0])
>>> y
array([0, 1, 0])
```

dask.array.**isinf**(*x*[, *out*])

Test element-wise for positive or negative infinity.

Returns a boolean array of the same shape as *x*, True where `x == +/-inf`, otherwise False.

Parameters **x** : array_like

Input values

out : array_like, optional

An array with the same shape as *x* to store the result.

Returns **y** : bool (scalar) or boolean ndarray

For scalar input, the result is a new boolean with value True if the input is positive or negative infinity; otherwise the value is False.

For array input, the result is a boolean array with the same shape as the input and the values are True where the corresponding element of the input is positive or negative infinity; elsewhere the values are False. If a second argument was supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True, respectively. The return value *y* is then a reference to that array.

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], dtype=bool)
```

```
>>> 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])
```

dask.array.**isnan**(*x*[, *out*])

Test element-wise for NaN and return result as a boolean array.

Parameters **x** : array_like

Input array.

Returns **y** : {ndarray, bool}

For scalar input, the result is a new boolean with value True if the input is NaN; otherwise the value is False.

For array input, the result is a boolean array of the same dimensions as the input and the values are True if the corresponding element of the input is NaN; otherwise the values are False.

See also:

isinf, isneginf, isposinf, *isfinite*

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], dtype=bool)
```

dask.array.**isnull**(*values*)
pandas.isnull for dask arrays

dask.array.**isreal**()
Returns a bool array, where True if input element is real.

If element has complex type with zero complex part, the return value for that element is True.

Parameters **x** : array_like

Input array.

Returns **out** : ndarray, bool

Boolean array of same shape as *x*.

See also:

iscomplex

isrealobj Return True if x is not a complex type.

Examples

```
>>> np.isreal([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([False,  True,  True,  True,  True, False], dtype=bool)
```

dask.array.**ldexp**(*x1*, *x2*[, *out*])
Returns x1 * 2**x2, element-wise.

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.

out : ndarray, optional

Output array for the result.

Returns **y** : ndarray or scalar

The result of `x1 * 2**x2`.

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=float32)
```

```
>>> x = np.arange(6)
>>> np.ldexp(*np.frexp(x))
array([ 0.,  1.,  2.,  3.,  4.,  5.])
```

`dask.array.`**`linspace`**(*start*, *stop*, *num=50*, *chunks=None*, *dtype=None*)

Return *num* evenly spaced values over the closed interval [*start*, *stop*].

TODO: implement the *endpoint*, *restep*, and *dtype* keyword args

Parameters **start** : scalar

The starting value of the sequence.

stop : scalar

The last value of the sequence.

num : int, optional

Number of samples to include in the returned dask array, including the endpoints.

chunks : int

The number of samples on each block. Note that the last block will have fewer samples if *num % blocksize != 0*

Returns **samples** : dask array

`dask.array.`**`log`**(*x*[, *out*])

Natural logarithm, element-wise.

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.

Returns **y** : ndarray

The natural logarithm of *x*, element-wise.

See also:

log10, *log2*, *log1p*, `emath.log`

Notes

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log* is a complex analytical function that has a branch cut *[-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.

References

[R87], *[R88]*

Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([  0.,   1.,   2., -Inf])
```

`dask.array.`**`log10`**(*x*[, *out*])

Return the base 10 logarithm of the input array, element-wise.

Parameters **x** : array_like

Input values.

Returns **y** : ndarray

The logarithm to the base 10 of *x*, element-wise. NaNs are returned where x is negative.

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.

References

[R89], *[R90]*

Examples

```
>>> np.log10([1e-15, -3.])
array([-15.,  NaN])
```

dask.array.**log1p**(*x*[, *out*])

Return the natural logarithm of one plus the input array, element-wise.

Calculates `log(1 + x)`.

Parameters **x** : array_like

Input values.

Returns **y** : ndarray

Natural logarithm of *1 + x*, element-wise.

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 *[-inf, -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

[R91], *[R92]*

Examples

```
>>> np.log1p(1e-99)
1e-99
>>> np.log(1 + 1e-99)
0.0
```

dask.array.**log2**(*x*[, *out*])

Base-2 logarithm of *x*.

Parameters **x** : array_like

Input values.

Returns **y** : ndarray

Base-2 logarithm of *x*.

See also:

log, *log10*, *log1p*, emath.log2

Notes

New in version 1.3.0.

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *2**z = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log2* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log2* is a complex analytical function that has a branch cut *[-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.

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])
```

dask.array.**logaddexp**(*x1*, *x2*[, *out*])

Logarithm of the sum of exponentiations of the inputs.

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.

Returns **result** : ndarray

Logarithm of `exp(x1) + exp(x2)`.

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)
3.5000000000000057e-50
```

dask.array.**logaddexp2**(*x1*, *x2*[, *out*])

Logarithm of the sum of exponentiations of the inputs in base-2.

Calculates `log2(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.

out : ndarray, optional

Array to store results in.

Returns **result** : ndarray

Base-2 logarithm of `2**x1 + 2**x2`.

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
```

dask.array.**logical_and**(*x1*, *x2*[, *out*])

Compute the truth value of x1 AND x2 element-wise.

Parameters **x1, x2** : array_like

Input arrays. *x1* and *x2* must be of the same shape.

Returns **y** : {ndarray, bool}

Boolean result with the same shape as *x1* and *x2* of the logical AND operation on corresponding elements of *x1* and *x2*.

See also:

logical_or, *logical_not*, *logical_xor*, bitwise_and

Examples

```
>>> np.logical_and(True, False)
False
>>> np.logical_and([True, False], [False, False])
array([False, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_and(x>1, x<4)
array([False, False,  True,  True, False], dtype=bool)
```

dask.array.**logical_not**(*x*[, *out*])

Compute the truth value of NOT x element-wise.

Parameters **x** : array_like

Logical NOT is applied to the elements of *x*.

Returns **y** : bool or ndarray of bool

Boolean result with the same shape as *x* of the NOT operation on elements of *x*.

See also:

logical_and, *logical_or*, *logical_xor*

Examples

```
>>> np.logical_not(3)
False
>>> np.logical_not([True, False, 0, 1])
array([False,  True,  True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_not(x<3)
array([False, False, False,  True,  True], dtype=bool)
```

dask.array.**logical_or**(*x1*, *x2*[, *out*])

Compute the truth value of x1 OR x2 element-wise.

Parameters **x1, x2** : array_like

Logical OR is applied to the elements of *x1* and *x2*. They have to be of the same shape.

Returns **y** : {ndarray, bool}

Boolean result with the same shape as *x1* and *x2* of the logical OR operation on elements of *x1* and *x2*.

See also:

logical_and, *logical_not*, *logical_xor*, bitwise_or

Examples

```
>>> np.logical_or(True, False)
True
>>> np.logical_or([True, False], [False, False])
array([ True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_or(x < 1, x > 3)
array([ True, False, False, False,  True], dtype=bool)
```

dask.array.**logical_xor**(*x1*, *x2*[, *out*])

Compute the truth value of x1 XOR x2, element-wise.

Parameters **x1, x2** : array_like

Logical XOR is applied to the elements of *x1* and *x2*. They must be broadcastable to the same shape.

Returns **y** : bool or ndarray of bool

Boolean result of the logical XOR operation applied to the elements of *x1* and *x2*; the shape is determined by whether or not broadcasting of one or both arrays was required.

See also:

logical_and, *logical_or*, *logical_not*, bitwise_xor

Examples

```
>>> np.logical_xor(True, False)
True
>>> np.logical_xor([True, True, False, False], [True, False, True, False])
array([False,  True,  True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_xor(x < 1, x > 3)
array([ True, False, False, False,  True], dtype=bool)
```

Simple example showing support of broadcasting

```
>>> np.logical_xor(0, np.eye(2))
array([[ True, False],
       [False,  True]], dtype=bool)
```

dask.array.**max**(*a*, *axis=None*, *keepdims=False*, *split_every=None*)

Return the maximum of an array or maximum along an axis.

Parameters **a** : array_like

Input data.

axis : int, optional

Axis along which to operate. By default, flattened input is used.

out : ndarray, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **amax** : ndarray or scalar

Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

See also:

`amin` The minimum value of an array along a given axis, propagating any NaNs.

`nanmax` The maximum value of an array along a given axis, ignoring any NaNs.

`maximum` Element-wise maximum of two arrays, propagating any NaNs.

`fmax` Element-wise maximum of two arrays, ignoring any NaNs.

`argmax` Return the indices of the maximum values.

nanmin, *minimum*, *fmin*

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding max value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmax.

Don't use *amax* for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `maximum(a[0], a[1])` is faster than `amax(a, axis=0)`.

Examples

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amax(a)           # Maximum of the flattened array
3
>>> np.amax(a, axis=0)   # Maxima along the first axis
array([2, 3])
>>> np.amax(a, axis=1)   # Maxima along the second axis
array([1, 3])
```

```
>>> b = np.arange(5, dtype=np.float)
>>> b[2] = np.NaN
>>> np.amax(b)
nan
>>> np.nanmax(b)
4.0
```

`dask.array.`**`maximum`**(*x1*, *x2*[, *out*])

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_like

The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

Returns **y** : {ndarray, scalar}

The maximum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

minimum Element-wise minimum of two arrays, propagates NaNs.

fmax Element-wise maximum of two arrays, ignores NaNs.

amax The maximum value of an array along a given axis, propagates NaNs.

nanmax The maximum value of an array along a given axis, ignores NaNs.

fmin, amin, *nanmin*

Notes

The maximum is equivalent to `np.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are nans, but it is faster and does proper broadcasting.

Examples

```
>>> np.maximum([2, 3, 4], [1, 5, 2])
array([2, 5, 4])
```

```
>>> np.maximum(np.eye(2), [0.5, 2]) # broadcasting
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.maximum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([ NaN,  NaN,  NaN])
>>> np.maximum(np.Inf, 1)
inf
```

dask.array.**mean**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **a** : array_like

Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : data-type, optional

Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **m** : ndarray, see dtype parameter above

If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See also:

`average` Weighted average

`std`, *`var`*, *`nanmean`*, *`nanstd`*, *`nanvar`*

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
```

In single precision, *mean* can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.546875
```

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

`dask.array.`**`min`**(*a*, *axis=None*, *keepdims=False*, *split_every=None*)

Return the minimum of an array or minimum along an axis.

Parameters **a** : array_like

Input data.

axis : int, optional

Axis along which to operate. By default, flattened input is used.

out : ndarray, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **amin** : ndarray or scalar

Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

See also:

amax The maximum value of an array along a given axis, propagating any NaNs.

nanmin The minimum value of an array along a given axis, ignoring any NaNs.

minimum Element-wise minimum of two arrays, propagating any NaNs.

fmin Element-wise minimum of two arrays, ignoring any NaNs.

argmin Return the indices of the minimum values.

nanmax, *maximum*, *fmax*

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding min value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmin.

Don't use *amin* for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `minimum(a[0], a[1])` is faster than `amin(a, axis=0)`.

Examples

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amin(a)           # Minimum of the flattened array
0
>>> np.amin(a, axis=0)   # Minima along the first axis
array([0, 1])
>>> np.amin(a, axis=1)   # Minima along the second axis
array([0, 2])
```

```
>>> b = np.arange(5, dtype=np.float)
>>> b[2] = np.NaN
>>> np.amin(b)
nan
```

```
>>> np.nanmin(b)
0.0
```

dask.array.**minimum**(*x1*, *x2*[, *out*])

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_like

The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

Returns **y** : {ndarray, scalar}

The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

maximum Element-wise maximum of two arrays, propagates NaNs.

fmin Element-wise minimum of two arrays, ignores NaNs.

amin The minimum value of an array along a given axis, propagates NaNs.

nanmin The minimum value of an array along a given axis, ignores NaNs.

fmax, amax, *nanmax*

Notes

The minimum is equivalent to `np.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

Examples

```
>>> np.minimum([2, 3, 4], [1, 5, 2])
array([1, 3, 2])
```

```
>>> np.minimum(np.eye(2), [0.5, 2]) # broadcasting
array([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

```
>>> np.minimum([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([ NaN,  NaN,  NaN])
>>> np.minimum(-np.Inf, 1)
-inf
```

dask.array.**modf**(*x*)

dask.array.**moment**(*a*, *order*, *axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)

dask.array.**nanargmax**(*x*, *axis=None*, *split_every=None*)

dask.array.**nanargmin**(*x*, *axis=None*, *split_every=None*)

dask.array.**nancumprod**(*x*, *axis*, *dtype=None*)

Return the cumulative product of array elements over a given axis treating Not a Numbers (NaNs) as one. The cumulative product does not change when NaNs are encountered and leading NaNs are replaced by ones.

Ones are returned for slices that are all-NaN or empty.

New in version 1.12.0.

Parameters **a** : array_like

Input array.

axis : int, optional

Axis along which the cumulative product is computed. By default the input is flattened.

dtype : dtype, optional

Type of the returned array, as well as of the accumulator in which the elements are multiplied. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

Returns **nancumprod** : ndarray

A new array holding the result is returned unless *out* is specified, in which case it is returned.

See also:

numpy.cumprod Cumulative product across array propagating NaNs.

isnan Show which elements are NaN.

Examples

```
>>> np.nancumprod(1)
array([1])
>>> np.nancumprod([1])
array([1])
>>> np.nancumprod([1, np.nan])
array([ 1.,  1.])
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nancumprod(a)
array([ 1.,  2.,  6.,  6.])
>>> np.nancumprod(a, axis=0)
array([[ 1.,  2.],
       [ 3.,  2.]])
>>> np.nancumprod(a, axis=1)
array([[ 1.,  2.],
       [ 3.,  3.]])
```

dask.array.**nancumsum**(*x*, *axis*, *dtype=None*)

Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero. The cumulative sum does not change when NaNs are encountered and leading NaNs are replaced by zeros.

Zeros are returned for slices that are all-NaN or empty.

New in version 1.12.0.

Parameters **a** : array_like

Input array.

axis : int, optional

Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.

dtype : dtype, optional

Type of the returned array and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See *doc.ufuncs* (Section "Output arguments") for more details.

Returns **nancumsum** : ndarray.

A new array holding the result is returned unless *out* is specified, in which it is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

See also:

numpy.cumsum Cumulative sum across array propagating NaNs.

isnan Show which elements are NaN.

Examples

```
>>> np.nancumsum(1)
array([1])
>>> np.nancumsum([1])
array([1])
>>> np.nancumsum([1, np.nan])
array([ 1.,  1.])
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nancumsum(a)
array([ 1.,  3.,  6.,  6.])
>>> np.nancumsum(a, axis=0)
array([[ 1.,  2.],
       [ 4.,  2.]])
>>> np.nancumsum(a, axis=1)
array([[ 1.,  3.],
       [ 3.,  3.]])
```

`dask.array.`**nanmax**(*a*, *axis=None*, *keepdims=False*, *split_every=None*)

Return the maximum of an array or maximum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and NaN is returned for that slice.

Parameters **a** : array_like

Array containing numbers whose maximum is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the maximum is computed. The default is to compute the maximum of the flattened array.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

New in version 1.8.0.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

New in version 1.8.0.

Returns **nanmax** : ndarray

An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, an ndarray scalar is returned. The same dtype as *a* is returned.

See also:

nanmin The minimum value of an array along a given axis, ignoring any NaNs.

amax The maximum value of an array along a given axis, propagating any NaNs.

fmax Element-wise maximum of two arrays, ignoring any NaNs.

maximum Element-wise maximum of two arrays, propagating any NaNs.

isnan Shows which elements are Not a Number (NaN).

isfinite Shows which elements are neither NaN nor infinity.

amin, *fmin*, *minimum*

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. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to np.max.

Examples

```
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmax(a)
3.0
>>> np.nanmax(a, axis=0)
array([ 3.,  2.])
>>> np.nanmax(a, axis=1)
array([ 2.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> np.nanmax([1, 2, np.nan, np.NINF])
2.0
>>> np.nanmax([1, 2, np.nan, np.inf])
inf
```

dask.array.**nanmean**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)

Compute the arithmetic mean along the specified axis, ignoring NaNs.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

For all-NaN slices, NaN is returned and a *RuntimeWarning* is raised.

New in version 1.8.0.

Parameters **a** : array_like

Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : data-type, optional

Type to use in computing the mean. For integer inputs, the default is *float64*; for inexact inputs, it is the same as the input dtype.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **m** : ndarray, see dtype parameter above

If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned. Nan is returned for slices that contain only NaNs.

See also:

average Weighted average

mean Arithmetic mean taken while not ignoring NaNs

var, *nanvar*

Notes

The arithmetic mean is the sum of the non-NaN elements along the axis divided by the number of non-NaN elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32*. Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanmean(a)
2.6666666666666665
>>> np.nanmean(a, axis=0)
array([ 2.,  4.])
>>> np.nanmean(a, axis=1)
array([ 1.,  3.5])
```

dask.array.**nanmin**(*a*, *axis=None*, *keepdims=False*, *split_every=None*)

Return minimum of an array or minimum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and Nan is returned for that slice.

Parameters **a** : array_like

Array containing numbers whose minimum is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the minimum is computed. The default is to compute the minimum of the flattened array.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

New in version 1.8.0.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

New in version 1.8.0.

Returns **nanmin** : ndarray

An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, an ndarray scalar is returned. The same dtype as *a* is returned.

See also:

nanmax The maximum value of an array along a given axis, ignoring any NaNs.

amin The minimum value of an array along a given axis, propagating any NaNs.

fmin Element-wise minimum of two arrays, ignoring any NaNs.

minimum Element-wise minimum of two arrays, propagating any NaNs.

isnan Shows which elements are Not a Number (NaN).

isfinite Shows which elements are neither NaN nor infinity.

amax, *fmax*, *maximum*

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. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to np.min.

Examples

```
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmin(a)
1.0
>>> np.nanmin(a, axis=0)
array([ 1.,  2.])
>>> np.nanmin(a, axis=1)
array([ 1.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> np.nanmin([1, 2, np.nan, np.inf])
1.0
>>> np.nanmin([1, 2, np.nan, np.NINF])
-inf
```

dask.array.**nanprod**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)
Return the product of array elements over a given axis treating Not a Numbers (NaNs) as zero.

One is returned for slices that are all-NaN or empty.

New in version 1.10.0.

Parameters **a** : array_like

Array containing numbers whose sum is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the product is computed. The default is to compute the product of the flattened array.

dtype : data-type, optional

The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the platform (u)intp. In that case, the default will be either (u)int32 or (u)int64 depending on whether the platform is 32 or 64 bits. For inexact inputs, dtype must be inexact.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`. If provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details. The casting of NaN to integer can yield unexpected results.

keepdims : bool, optional

If True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **y** : ndarray or numpy scalar

See also:

numpy.prod Product across array propagating NaNs.

isnan Show which elements are NaN.

Notes

Numpy integer arithmetic is modular. If the size of a product exceeds the size of an integer accumulator, its value will wrap around and the result will be incorrect. Specifying `dtype=double` can alleviate that problem.

Examples

```
>>> np.nanprod(1)
1
>>> np.nanprod([1])
1
>>> np.nanprod([1, np.nan])
1.0
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanprod(a)
6.0
>>> np.nanprod(a, axis=0)
array([ 3.,  2.])
```

dask.array.**nanstd**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)

Compute the standard deviation along the specified axis, while ignoring NaNs.

Returns the standard deviation, a measure of the spread of a distribution, of the non-NaN array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a *RuntimeWarning* is raised.

New in version 1.8.0.

Parameters **a** : array_like

Calculate the standard deviation of the non-NaN values.

axis : int, optional

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

dtype : dtype, optional

Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

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 calculated values) will be cast if necessary.

ddof : int, optional

Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of non-NaN elements. By default *ddof* is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **standard_deviation** : ndarray, see dtype parameter above.

If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output array. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

See also:

var, *mean*, *std*, *nanvar*, *nanmean*

numpy.doc.ufuncs Section "Output arguments"

Notes

The standard deviation is the square root of the average of the squared deviations from the mean: `std = sqrt(mean(abs(x - x.mean())**2))`.

The average squared deviation is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, *std* takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the *dtype* keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanstd(a)
1.247219128924647
>>> np.nanstd(a, axis=0)
array([ 1.,  0.])
>>> np.nanstd(a, axis=1)
array([ 0.,  0.5])
```

dask.array.**nansum**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)

Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.

FutureWarning: In Numpy versions <= 1.8 Nan is returned for slices that are all-NaN or empty. In later versions zero will be returned.

Parameters **a** : array_like

Array containing numbers whose sum is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the sum is computed. The default is to compute the sum of the flattened array.

dtype : data-type, optional

The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the platform (u)intp. In that case, the default will be either (u)int32 or (u)int64 depending on whether the platform is 32 or 64 bits. For inexact inputs, dtype must be inexact.

New in version 1.8.0.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`. If provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details. The casting of NaN to integer can yield unexpected results.

New in version 1.8.0.

keepdims : bool, optional

If True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

New in version 1.8.0.

Returns **y** : ndarray or numpy scalar

See also:

`numpy.sum` Sum across array propagating NaNs.

`isnan` Show which elements are NaN.

`isfinite` Show which elements are not NaN or +/-inf.

Notes

If both positive and negative infinity are present, the sum will be Not A Number (NaN).

Numpy integer arithmetic is modular. If the size of a sum exceeds the size of an integer accumulator, its value will wrap around and the result will be incorrect. Specifying `dtype=double` can alleviate that problem.

Examples

```
>>> np.nansum(1)
1
>>> np.nansum([1])
1
>>> np.nansum([1, np.nan])
1.0
>>> a = np.array([[1, 1], [1, np.nan]])
>>> np.nansum(a)
3.0
>>> np.nansum(a, axis=0)
array([ 2.,  1.])
>>> np.nansum([1, np.nan, np.inf])
inf
```

```
    >>> np.nansum([1, np.nan, np.NINF])
    -inf
    >>> np.nansum([1, np.nan, np.inf, -np.inf]) # both +/- infinity present
    nan
```

dask.array.**nanvar**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)

Compute the variance along the specified axis, while ignoring NaNs.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a *RuntimeWarning* is raised.

New in version 1.8.0.

Parameters **a** : array_like

Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : data-type, optional

Type to use in computing the variance. For arrays of integer type the default is *float32*; for arrays of float types it is the same as the array type.

out : ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddof : int, optional

"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of non-NaN elements. By default *ddof* is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **variance** : ndarray, see dtype parameter above

If *out* is None, return a new array containing the variance, otherwise return a reference to the output array. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

See also:

std Standard deviation

mean Average

var Variance while not ignoring NaNs

nanstd, *nanmean*

numpy.doc.ufuncs Section "Output arguments"

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.var(a)
1.5555555555555554
>>> np.nanvar(a, axis=0)
array([ 1.,  0.])
>>> np.nanvar(a, axis=1)
array([ 0.,  0.25])
```

`dask.array.`**`nextafter`**(*x1*, *x2*[, *out*])

Return the next floating-point value after x1 towards x2, element-wise.

Parameters **x1** : array_like

Values to find the next representable value of.

x2 : array_like

The direction where to look for the next representable value of *x1*.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See *doc.ufuncs*.

Returns **out** : array_like

The next representable values of *x1* in the direction of *x2*.

Examples

```
>>> eps = np.finfo(np.float64).eps
>>> np.nextafter(1, 2) == eps + 1
True
>>> np.nextafter([1, 2], [2, 1]) == [eps + 1, 2 - eps]
array([ True,  True], dtype=bool)
```

`dask.array.`**`notnull`**(*values*)

pandas.notnull for dask arrays

dask.array.**ones**()
Blocked variant of ones

Follows the signature of ones exactly except that it also requires a keyword argument chunks=(...)

Original signature follows below.

Return a new array of given shape and type, filled with ones.

Parameters **shape** : int or sequence of ints

Shape of the new array, e.g., `(2, 3)` or `2`.

dtype : data-type, optional

The 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- or Fortran-contiguous (row- or column-wise) order in memory.

Returns **out** : ndarray

Array of ones with the given shape, dtype, and order.

See also:

zeros, ones_like

Examples

```
>>> np.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
```

```
>>> np.ones((5,), dtype=np.int)
array([1, 1, 1, 1, 1])
```

```
>>> np.ones((2, 1))
array([[ 1.],
       [ 1.]])
```

```
>>> s = (2,2)
>>> np.ones(s)
array([[ 1.,  1.],
       [ 1.,  1.]])
```

dask.array.**percentile**(*a*, *q*, *interpolation='linear'*)
Approximate percentile of 1-D array

See numpy.percentile for more information

dask.array.**prod**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)
Return the product of array elements over a given axis.

Parameters **a** : array_like

Input data.

axis : None or int or tuple of ints, optional

Axis or axes along which a product is performed. The default (*axis* = *None*) is perform a product over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a product is performed on multiple axes, instead of a single axis or all the axes as before.

dtype : data-type, optional

The data-type of the returned array, as well as of the accumulator in which the elements are multiplied. By default, if *a* is of integer type, *dtype* is the default platform integer. (Note: if the type of *a* is unsigned, then so is *dtype*.) Otherwise, the dtype is the same as that of *a*.

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.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **product_along_axis** : ndarray, see *dtype* parameter above.

An array shaped as *a* but with the specified axis removed. Returns a reference to *out* if specified.

See also:

`ndarray.prod` equivalent method

`numpy.doc.ufuncs` Section "Output arguments"

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

```
>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x) #random
16
```

Examples

By default, calculate the product of all elements:

```
>>> np.prod([1.,2.])
2.0
```

Even when the input array is two-dimensional:

```
>>> np.prod([[1.,2.],[3.,4.]])
24.0
```

But we can also specify the axis over which to multiply:

```
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([  2.,  12.])
```

If the type of *x* is unsigned, then the output type is the unsigned platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True
```

If *x* is of a signed integer type, then the output type is the default platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == np.int
True
```

dask.array.**rad2deg**(*x*[, *out*])

Convert angles from radians to degrees.

Parameters **x** : array_like

Angle in radians.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns **y** : ndarray

The corresponding angle in degrees.

See also:

deg2rad Convert angles from degrees to radians.

unwrap Remove large jumps in angle by wrapping.

Notes

New in version 1.3.0.

rad2deg(x) is `180 * x / pi`.

Examples

```
>>> np.rad2deg(np.pi/2)
90.0
```

dask.array.**radians**(*x*[, *out*])

Convert angles from degrees to radians.

Parameters **x** : array_like

Input array in degrees.

out : ndarray, optional

Output array of same shape as *x*.

Returns **y** : ndarray

The corresponding radian values.

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
```

dask.array.**ravel**(*array*)

Return a flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Parameters **a** : array_like

Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.

order : {'C','F', 'A', 'K'}, optional

The elements of *a* are read using this index order. 'C' means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

Returns **1d_array** : ndarray

Output of the same dtype as *a*, and of shape `(a.size,)`.

See also:

ndarray.flat 1-D iterator over an array.

ndarray.flatten 1-D array copy of the elements of an array in row-major order.

Notes

In C-like (row-major) order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for Fortran-like, or column-major, index ordering.

Examples

It is equivalent to `reshape(-1, order=order)`.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print np.ravel(x)
[1 2 3 4 5 6]
```

```
>>> print x.reshape(-1)
[1 2 3 4 5 6]
```

```
>>> print np.ravel(x, order='F')
[1 4 2 5 3 6]
```

When `order` is 'A', it will preserve the array's 'C' or 'F' ordering:

```
>>> print np.ravel(x.T)
[1 4 2 5 3 6]
>>> print np.ravel(x.T, order='A')
[1 2 3 4 5 6]
```

When `order` is 'K', it will preserve orderings that are neither 'C' nor 'F', but won't reverse axes:

```
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])
```

```
>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[ 0,  2,  4],
        [ 1,  3,  5]],
       [[ 6,  8, 10],
        [ 7,  9, 11]]])
>>> a.ravel(order='C')
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
>>> a.ravel(order='K')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

`dask.array.`**`real`**(**args*, ***kwargs*)

Return the real part of the elements of the array.

Parameters **val** : array_like

Input array.

Returns **out** : ndarray

Output array. 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])
```

dask.array.**rechunk**(*x*, *chunks*)

Convert blocks in dask array x for new chunks.

```
    >>> import dask.array as da
    >>> a = np.random.uniform(0, 1, 7**4).reshape((7,) * 4)
    >>> x = da.from_array(a, chunks=((2, 3, 2),)*4)
    >>> x.chunks
    ((2, 3, 2), (2, 3, 2), (2, 3, 2), (2, 3, 2))
```

```
    >>> y = rechunk(x, chunks=((2, 4, 1), (4, 2, 1), (4, 3), (7,)))
    >>> y.chunks
    ((2, 4, 1), (4, 2, 1), (4, 3), (7,))
```

chunks also accept dict arguments mapping axis to blockshape

```
    >>> y = rechunk(x, chunks={1: 2})  # rechunk axis 1 with blockshape 2
```

Parameters **x: dask array**

chunks: the new block dimensions to create

dask.array.**reshape**(*array*, *shape*)

Gives a new shape to an array without changing its data.

Parameters **a** : array_like

Array to be reshaped.

newshape : int or tuple of ints

The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

order : {'C', 'F', 'A'}, optional

Read the elements of *a* using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise.

Returns **reshaped_array** : ndarray

This will be a new view object if possible; otherwise, it will be a copy. Note there is no guarantee of the *memory layout* (C- or Fortran- contiguous) of the returned array.

See also:

ndarray.reshape Equivalent method.

Notes

It is not always possible to change the shape of an array without copying the data. If you want an error to be raise if the data is copied, you should assign the new shape to the shape attribute of the array:

```
>>> a = np.zeros((10, 2))
# A transpose make the array non-contiguous
>>> b = a.T
# Taking a view makes it possible to modify the shape without modifying the
# initial object.
>>> c = b.view()
>>> c.shape = (20)
AttributeError: incompatible shape for a non-contiguous array
```

The *order* keyword gives the index ordering both for *fetching* the values from *a*, and then *placing* the values into the output array. For example, let's say you have an array:

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

You can think of reshaping as first raveling the array (using the given index order), then inserting the elements from the raveled array into the new array using the same kind of index ordering as was used for the raveling.

```
>>> np.reshape(a, (2, 3)) # C-like index ordering
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(np.ravel(a), (2, 3)) # equivalent to C ravel then C reshape
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(a, (2, 3), order='F') # Fortran-like index ordering
array([[0, 4, 3],
       [2, 1, 5]])
>>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
array([[0, 4, 3],
       [2, 1, 5]])
```

Examples

```
>>> a = np.array([[1,2,3], [4,5,6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])
```

```
>>> np.reshape(a, (3,-1))       # the unspecified value is inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

`dask.array.rint`(*x*[, *out*])

Round elements of the array to the nearest integer.

Parameters **x** : array_like

Input array.

Returns **out** : {ndarray, scalar}

Output array is same shape and type as *x*.

See also:

ceil, *floor*, *trunc*

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.rint(a)
array([-2., -2., -0.,  0.,  2.,  2.,  2.])
```

dask.array.**sign**(*x*[, *out*])

Returns an element-wise indication of the sign of a number.

The *sign* function returns `-1 if x < 0, 0 if x==0, 1 if x > 0`.

Parameters **x** : array_like

Input values.

Returns **y** : ndarray

The sign of *x*.

Examples

```
>>> np.sign([-5., 4.5])
array([-1.,  1.])
>>> np.sign(0)
0
```

dask.array.**signbit**(*x*[, *out*])

Returns element-wise True where signbit is set (less than zero).

Parameters **x** : array_like

The input value(s).

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See *doc.ufuncs*.

Returns **result** : ndarray of bool

Output array, or reference to *out* if that was supplied.

Examples

```
>>> np.signbit(-1.2)
True
>>> np.signbit(np.array([1, -2.3, 2.1]))
array([False,  True, False], dtype=bool)
```

dask.array.**sin**(*x*[, *out*])

Trigonometric sine, element-wise.

Parameters **x** : array_like

Angle, in radians (2π rad equals 360 degrees).

Returns **y** : array_like

The sine of each element of x.

See also:

arcsin, *sinh*, *cos*

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Examples

Print sine of one angle:

```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Plot the sine function:

```
>>> import matplotlib.pylab 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()
```

`dask.array.`**`sinh`**(*x*[, *out*])

Hyperbolic sine, element-wise.

Equivalent to `1/2 * (np.exp(x) - np.exp(-x))` or `-1j * np.sin(1j*x)`.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Output array of same shape as *x*.

Returns **y** : ndarray

The corresponding hyperbolic sine values.

Raises **ValueError: invalid return array shape**

if *out* is provided and *out.shape* != *x.shape* (See Examples)

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
>>> 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: invalid return array shape
```

dask.array.**sqrt**(*x*[, *out*])

Return the positive square-root of an array, element-wise.

Parameters **x** : array_like

The values whose square-roots are required.

out : ndarray, optional

Alternate array object in which to put the result; if provided, it must have the same shape as *x*

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.

See also:

lib.scimath.sqrt A version which returns complex numbers when given negative reals.

Notes

sqrt has–consistent with common convention–as its branch cut the real "interval" [-*inf*, 0), and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

Examples

```
>>> np.sqrt([1,4,9])
array([ 1.,  2.,  3.])
```

```
>>> np.sqrt([4, -1, -3+4J])
array([ 2.+0.j,  0.+1.j,  1.+2.j])
```

```
>>> np.sqrt([4, -1, numpy.inf])
array([  2.,  NaN,  Inf])
```

dask.array.**square**(*x*[, *out*])

Return the element-wise square of the input.

Parameters **x** : array_like

Input data.

Returns **out** : ndarray

Element-wise *x***x*, of the same shape and dtype as *x*. Returns scalar if *x* is a scalar.

See also:

numpy.linalg.matrix_power, *sqrt*, power

Examples

```
>>> np.square([-1j, 1])
array([-1.-0.j,  1.+0.j])
```

dask.array.**squeeze**(*a*, *axis=None*)

Remove single-dimensional entries from the shape of an array.

Parameters **a** : array_like

Input data.

axis : None or int or tuple of ints, optional

New in version 1.7.0.

Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

Returns **squeezed** : ndarray

The input array, but with with all or a subset of the dimensions of length 1 removed. This is always *a* itself or a view into *a*.

Examples

```
>>> x = np.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=(2,)).shape
(1, 3)
```

dask.array.**stack**(*seq*, *axis=0*)

Stack arrays along a new axis

Given a sequence of dask Arrays form a new dask Array by stacking them along a new dimension (axis=0 by default)

See also:

concatenate

Examples

Create slices

```
>>> import dask.array as da
>>> import numpy as np
```

```
>>> data = [from_array(np.ones((4, 4)), chunks=(2, 2))
...          for i in range(3)]
```

```
>>> x = da.stack(data, axis=0)
>>> x.shape
(3, 4, 4)
```

```
>>> da.stack(data, axis=1).shape
(4, 3, 4)
```

```
>>> da.stack(data, axis=-1).shape
(4, 4, 3)
```

Result is a new dask Array

dask.array.**std**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters **a** : array_like

Calculate the standard deviation of these values.

axis : int, optional

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

dtype : dtype, optional

Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

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 calculated values) will be cast if necessary.

ddof : int, optional

Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **standard_deviation** : ndarray, see dtype parameter above.

If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See also:

var, *mean*, *nanmean*, *nanstd*, *nanvar*

numpy.doc.ufuncs Section "Output arguments"

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `std = sqrt(mean(abs(x - x.mean())**2))`.

The average squared deviation is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, *std* takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the *dtype* keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1.,  1.])
>>> np.std(a, axis=1)
array([ 0.5,  0.5])
```

In single precision, std() can be inaccurate:

```
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

dask.array.**sum**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)

Sum of array elements over a given axis.

Parameters **a** : array_like

Elements to sum.

axis : None or int or tuple of ints, optional

Axis or axes along which a sum is performed. The default (*axis* = *None*) is perform a sum over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a sum is performed on multiple axes, instead of a single axis or all the axes as before.

dtype : dtype, optional

The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the default platform integer. In that case, the default platform integer is used instead.

out : ndarray, optional

Array into which the output is placed. By default, a new array is created. If *out* is given, it must be of the appropriate shape (the shape of *a* with *axis* removed, i.e., `numpy.delete(a.shape, axis)`). Its type is preserved. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **sum_along_axis** : ndarray

An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

See also:

ndarray.sum Equivalent method.

cumsum Cumulative sum of array elements.

trapz Integration of array values using the composite trapezoidal rule.

mean, average

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
```

If the accumulator is too small, overflow occurs:

```
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

dask.array.**take**(*a*, *indices*, *axis=0*)

Take elements from an array along an axis.

This function does the same thing as "fancy" indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis.

Parameters **a** : array_like

The source array.

indices : array_like

The indices of the values to extract.

New in version 1.8.0.

Also allow scalars for indices.

axis : int, optional

The axis over which to select values. By default, the flattened input array is used.

out : ndarray, optional

If provided, the result will be placed in this array. It should be of the appropriate shape and dtype.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave.

- 'raise' – raise an error (default)
- 'wrap' – wrap around
- 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

Returns **subarray** : ndarray

The returned array has the same type as *a*.

See also:

`ndarray.take` equivalent method

Examples

```
>>> a = [4, 3, 5, 7, 6, 8]
>>> indices = [0, 1, 4]
>>> np.take(a, indices)
array([4, 3, 6])
```

In this example if *a* is an ndarray, "fancy" indexing can be used.

```
>>> a = np.array(a)
>>> a[indices]
array([4, 3, 6])
```

If *indices* is not one dimensional, the output also has these dimensions.

```
>>> np.take(a, [[0, 1], [2, 3]])
array([[4, 3],
       [5, 7]])
```

`dask.array.tan`(*x*[, *out*])

Compute tangent element-wise.

Equivalent to `np.sin(x)/np.cos(x)` element-wise.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Output array of same shape as *x*.

Returns **y** : ndarray

The corresponding tangent values.

Raises **ValueError: invalid return array shape**

if *out* is provided and *out.shape* != *x.shape* (See Examples)

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
>>> 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: invalid return array shape
```

dask.array.**tanh**(*x*[, *out*])

Compute hyperbolic tangent element-wise.

Equivalent to `np.sinh(x)/np.cosh(x)` or `-1j * np.tan(1j*x)`.

Parameters **x** : array_like

Input array.

out : ndarray, optional

Output array of same shape as *x*.

Returns **y** : ndarray

The corresponding hyperbolic tangent values.

Raises **ValueError: invalid return array shape**

if *out* is provided and *out.shape* != *x.shape* (See Examples)

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

[R93], *[R94]*

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
>>> out2 = np.tanh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.tanh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

dask.array.**tensordot**(*lhs*, *rhs*, *axes=2*)

Compute tensor dot product along specified axes for arrays >= 1-D.

Given two tensors (arrays of dimension greater than or equal to one), *a* and *b*, and an array_like object containing two array_like objects, `(a_axes, b_axes)`, sum the products of *a*'s and *b*'s elements (components) over the axes specified by `a_axes` and `b_axes`. The third argument can be a single non-negative integer_like scalar, `N`; if it is such, then the last `N` dimensions of *a* and the first `N` dimensions of *b* are summed over.

Parameters **a, b** : array_like, len(shape) >= 1

Tensors to "dot".

axes : variable type

- integer_like scalar Number of axes to sum over (applies to both arrays); or
- (2,) array_like, both elements array_like of the same length List of axes to be summed over, first sequence applying to *a*, second to *b*.

See also:

dot, `einsum`

Notes

When there is more than one axis to sum over - and they are not the last (first) axes of *a* (*b*) - the argument *axes* should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth.

Examples

A "traditional" example:

```
>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> c = np.tensordot(a,b, axes=([1,0],[0,1]))
>>> c.shape
(5, 2)
>>> c
array([[ 4400.,  4730.],
       [ 4532.,  4874.],
       [ 4664.,  5018.],
       [ 4796.,  5162.],
       [ 4928.,  5306.]])
>>> # A slower but equivalent way of computing the same...
>>> d = np.zeros((5,2))
>>> for i in range(5):
...   for j in range(2):
...     for k in range(3):
...       for n in range(4):
...         d[i,j] += a[k,n,i] * b[n,k,j]
>>> c == d
```

```
array([[ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True]], dtype=bool)
```

An extended example taking advantage of the overloading of + and *:

```
>>> a = np.array(range(1, 9))
>>> a.shape = (2, 2, 2)
>>> A = np.array(('a', 'b', 'c', 'd'), dtype=object)
>>> A.shape = (2, 2)
>>> a; A
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
array([[a, b],
       [c, d]], dtype=object)
```

```
>>> np.tensordot(a, A) # third argument default is 2
array([abbcccdddd, aaaaabbbbbbccccccdddddddd], dtype=object)
```

```
>>> np.tensordot(a, A, 1)
array([[[acc, bdd],
        [aaacccc, bbbdddd]],
       [[aaaaacccccc, bbbbbddddddd],
        [aaaaaaaccccccc, bbbbbbbddddddddd]]], dtype=object)
```

```
>>> np.tensordot(a, A, 0) # "Left for reader" (result too long to incl.)
array([[[[[a, b],
          [c, d]],
          ...
```

```
>>> np.tensordot(a, A, (0, 1))
array([[[abbbbb, cddddd],
        [aabbbbbb, ccdddddd]],
       [[aaabbbbbbb, cccddddddd],
        [aaaabbbbbbbb, ccccddddddddd]]], dtype=object)
```

```
>>> np.tensordot(a, A, (2, 1))
array([[[abb, cdd],
        [aaabbbb, cccdddd]],
       [[aaaaabbbbbb, ccccccddddddd],
        [aaaaaaabbbbbbbb, cccccccddddddddd]]], dtype=object)
```

```
>>> np.tensordot(a, A, ((0, 1), (0, 1)))
array([abbbcccccddddddd, aabbbbccccccddddddddd], dtype=object)
```

```
>>> np.tensordot(a, A, ((2, 1), (1, 0)))
array([acccbbdddd, aaaaaccccccbbbbbbddddddddd], dtype=object)
```

`dask.array.`**`topk`**(*k*, *x*)

The top k elements of an array

Returns the k greatest elements of the array in sorted order. Only works on arrays of a single dimension.

```
>>> x = np.array([5, 1, 3, 6])
>>> d = from_array(x, chunks=2)
>>> d.topk(2).compute()
array([6, 5])
```

Runs in near linear time, returns all results in a single chunk so all k elements must fit in memory.

dask.array.**transpose**(*a*, *axes=None*)

Permute the dimensions of an array.

Parameters **a** : array_like

Input array.

axes : list of ints, optional

By default, reverse the dimensions, otherwise permute the axes according to the values given.

Returns **p** : ndarray

a with its axes permuted. A view is returned whenever possible.

See also:

`rollaxis`

Examples

```
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])
```

```
>>> np.transpose(x)
array([[0, 2],
       [1, 3]])
```

```
>>> x = np.ones((1, 2, 3))
>>> np.transpose(x, (1, 0, 2)).shape
(2, 1, 3)
```

dask.array.**tril**(*m*, *k=0*)

Lower triangle of an array with elements above the *k*-th diagonal zeroed.

Parameters **m** : array_like, shape (M, M)

Input array.

k : int, optional

Diagonal above which to zero elements. $k = 0$ (the default) is the main diagonal, $k < 0$ is below it and $k > 0$ is above.

Returns **tril** : ndarray, shape (M, M)

Lower triangle of *m*, of same shape and data-type as *m*.

See also:

triu upper triangle of an array

dask.array.**triu**(*m*, *k=0*)

Upper triangle of an array with elements above the *k*-th diagonal zeroed.

Parameters **m** : array_like, shape (M, N)

Input array.

k : int, optional

Diagonal above which to zero elements. *k = 0* (the default) is the main diagonal, *k < 0* is below it and *k > 0* is above.

Returns **triu** : ndarray, shape (M, N)

Upper triangle of *m*, of same shape and data-type as *m*.

See also:

tril lower triangle of an array

dask.array.**trunc**(*x*[, *out*])

Return the truncated value of the input, element-wise.

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.

Returns **y** : {ndarray, scalar}

The truncated value of each element in *x*.

See also:

ceil, *floor*, *rint*

Notes

New in version 1.3.0.

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.trunc(a)
array([-1., -1., -0.,  0.,  1.,  1.,  2.])
```

dask.array.**unique**(*x*)

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are two optional outputs in addition to the unique elements: the indices of the input array that give the unique values, and the indices of the unique array that reconstruct the input array.

Parameters **ar** : array_like

Input array. This will be flattened if it is not already 1-D.

return_index : bool, optional

If True, also return the indices of *ar* that result in the unique array.

return_inverse : bool, optional

If True, also return the indices of the unique array that can be used to reconstruct *ar*.

Returns **unique** : ndarray

The sorted unique values.

unique_indices : ndarray, optional

The indices of the first occurrences of the unique values in the (flattened) original array. Only provided if *return_index* is True.

unique_inverse : ndarray, optional

The indices to reconstruct the (flattened) original array from the unique array. Only provided if *return_inverse* is True.

See also:

numpy.lib.arraysetops Module with a number of other functions for performing set operations on arrays.

Examples

```
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the indices of the original array that give the unique values:

```
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'],
      dtype='|S1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'],
      dtype='|S1')
```

Reconstruct the input array from the unique values:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
>>> u[indices]
array([1, 2, 6, 4, 2, 3, 2])
```

dask.array.**var**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)
Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters **a** : array_like

Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : data-type, optional

Type to use in computing the variance. For arrays of integer type the default is *float32*; for arrays of float types it is the same as the array type.

out : ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddof : int, optional

"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **variance** : ndarray, see dtype parameter above

If `out=None`, returns a new array containing the variance; otherwise, a reference to the output array is returned.

See also:

std, *mean*, *nanmean*, *nanstd*, *nanvar*

numpy.doc.ufuncs Section "Output arguments"

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([ 1.,  1.])
>>> np.var(a, axis=1)
array([ 0.25,  0.25])
```

In single precision, var() can be inaccurate:

```
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096
```

Computing the variance in float64 is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

dask.array.**vnorm**(*a*, *ord=None*, *axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)
Vector norm

See np.linalg.norm

dask.array.**vstack**(*tup*)
Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by *vsplit*.

Parameters **tup** : sequence of ndarrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns **stacked** : ndarray

The array formed by stacking the given arrays.

See also:

hstack Stack arrays in sequence horizontally (column wise).

dstack Stack arrays in sequence depth wise (along third dimension).

concatenate Join a sequence of arrays together.

vsplit Split array into a list of multiple sub-arrays vertically.

Notes

Equivalent to `np.concatenate(tup, axis=0)` if *tup* contains arrays that are at least 2-dimensional.

Examples

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

`dask.array.`**`where`**(*condition*[, *x*, *y*])

Return elements, either from *x* or *y*, depending on *condition*.

If only *condition* is given, return `condition.nonzero()`.

Parameters **condition** : array_like, bool

When True, yield *x*, otherwise yield *y*.

x, y : array_like, optional

Values from which to choose. *x* and *y* need to have the same shape as *condition*.

Returns **out** : ndarray or tuple of ndarrays

If both *x* and *y* are specified, the output array contains elements of *x* where *condition* is True, and elements from *y* elsewhere.

If only *condition* is given, return the tuple `condition.nonzero()`, the indices where *condition* is True.

See also:

`nonzero`, *`choose`*

Notes

If *x* and *y* are given and input arrays are 1-D, *where* is equivalent to:

```
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]
```

Examples

```
>>> np.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...          [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])
```

```
>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))
```

```
>>> x = np.arange(9.).reshape(3, 3)
>>> np.where( x > 5 )
(array([2, 2, 2]), array([0, 1, 2]))
>>> x[np.where( x > 3.0 )]               # Note: result is 1D.
array([ 4.,  5.,  6.,  7.,  8.])
>>> np.where(x < 5, x, -1)               # Note: broadcasting.
array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])
```

Find the indices of elements of *x* that are in *goodvalues*.

```
>>> goodvalues = [3, 4, 7]
>>> ix = np.in1d(x.ravel(), goodvalues).reshape(x.shape)
>>> ix
array([[False, False, False],
       [ True,  True, False],
       [False,  True, False]], dtype=bool)
>>> np.where(ix)
(array([1, 1, 2]), array([0, 1, 1]))
```

dask.array.**zeros**()

Blocked variant of zeros

Follows the signature of zeros exactly except that it also requires a keyword argument chunks=(...)

Original signature follows below. zeros(shape, dtype=float, order='C')

Return a new array of given shape and type, filled with zeros.

Parameters **shape** : int or sequence of ints

Shape of the new array, e.g., `(2, 3)` or `2`.

dtype : data-type, optional

The 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- or Fortran-contiguous (row- or column-wise) order in memory.

Returns **out** : ndarray

Array of zeros with the given shape, dtype, and order.

See also:

zeros_like Return an array of zeros with shape and type of input.

ones_like Return an array of ones with shape and type of input.

empty_like Return an empty array with shape and type of input.

ones Return a new array setting values to one.

empty Return a new uninitialized array.

Examples

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

```
>>> np.zeros((5,), dtype=numpy.int)
array([0, 0, 0, 0, 0])
```

```
>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])
```

```
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

dask.array.linalg.**cholesky**(*a*, *lower=False*)

Returns the Cholesky decomposition, $A = LL^*$ or $A = U^*U$ of a Hermitian positive-definite matrix A.

Parameters **a** : (M, M) array_like

Matrix to be decomposed

lower : bool, optional

Whether to compute the upper or lower triangular Cholesky factorization. Default is upper-triangular.

Returns **c** : (M, M) Array

Upper- or lower-triangular Cholesky factor of *a*.

dask.array.linalg.**inv**(*a*)

Compute the inverse of a matrix with LU decomposition and forward / backward substitutions.

Parameters **a** : array_like

Square matrix to be inverted.

Returns **ainv** : Array

Inverse of the matrix *a*.

dask.array.linalg.**lstsq**(*a*, *b*)

Return the least-squares solution to a linear matrix equation using QR decomposition.

Solves the equation *a x = b* by computing a vector *x* that minimizes the Euclidean 2-norm || *b - a x* ||^2. The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of *a* can be less than, equal to, or greater than its number of linearly independent columns). If *a* is square and of full rank, then *x* (but for round-off error) is the "exact" solution of the equation.

Parameters **a** : (M, N) array_like

"Coefficient" matrix.

b : (M,) array_like

Ordinate or "dependent variable" values.

Returns **x** : (N,) Array

Least-squares solution. If *b* is two-dimensional, the solutions are in the *K* columns of *x*.

residuals : (1,) Array

Sums of residuals; squared Euclidean 2-norm for each column in `b - a*x`.

rank : Array

Rank of matrix *a*.

s : (min(M, N),) Array

Singular values of *a*.

`dask.array.linalg.`**`lu`**(*a*)

Compute the lu decomposition of a matrix.

Returns p: Array, permutation matrix

l: Array, lower triangular matrix with unit diagonal.

u: Array, upper triangular matrix

Examples

```
>>> p, l, u = da.linalg.lu(x)
```

`dask.array.linalg.`**`qr`**(*a*, *name=None*)

Compute the qr factorization of a matrix.

Returns q: Array, orthonormal

r: Array, upper-triangular

See also:

`np.linalg.qr` Equivalent NumPy Operation

`dask.array.linalg.tsqr` Actual implementation with citation

Examples

```
>>> q, r = da.linalg.qr(x)
```

`dask.array.linalg.`**`solve`**(*a*, *b*, *sym_pos=False*)

Solve the equation `a x = b` for `x`. By default, use LU decomposition and forward / backward substitutions. When `sym_pos` is `True`, use Cholesky decomposition.

Parameters **a** : (M, M) array_like

A square matrix.

b : (M,) or (M, N) array_like

Right-hand side matrix in `a x = b`.

sym_pos : bool

Assume a is symmetric and positive definite. If `True`, use Cholesky decomposition.

Returns **x** : (M,) or (M, N) Array

Solution to the system `a x = b`. Shape of the return matches the shape of *b*.

`dask.array.linalg.`**`solve_triangular`**(*a*, *b*, *lower=False*)

Solve the equation *a x = b* for *x*, assuming a is a triangular matrix.

Parameters **a** : (M, M) array_like

A triangular matrix

b : (M,) or (M, N) array_like

Right-hand side matrix in *a x = b*

lower : bool, optional

Use only data contained in the lower triangle of *a*. Default is to use upper triangle.

Returns **x** : (M,) or (M, N) array

Solution to the system *a x = b*. Shape of return matches *b*.

`dask.array.linalg.`**`svd`**(*a*, *name=None*)

Compute the singular value decomposition of a matrix.

Returns u: Array, unitary / orthogonal

s: Array, singular values in decreasing order (largest first)

v: Array, unitary / orthogonal

See also:

`np.linalg.svd` Equivalent NumPy Operation

`dask.array.linalg.tsqr` Actual implementation with citation

Examples

```
>>> u, s, v = da.linalg.svd(x)
```

`dask.array.linalg.`**`svd_compressed`**(*a*, *k*, *n_power_iter=0*, *seed=None*, *name=None*)

Randomly compressed rank-k thin Singular Value Decomposition.

This computes the approximate singular value decomposition of a large array. This algorithm is generally faster than the normal algorithm but does not provide exact results. One can balance between performance and accuracy with input parameters (see below).

Parameters **a: Array**

Input array

k: int

Rank of the desired thin SVD decomposition.

n_power_iter: int

Number of power iterations, useful when the singular values decay slowly. Error decreases exponentially as n_power_iter increases. In practice, set n_power_iter <= 4.

Returns u: Array, unitary / orthogonal

s: Array, singular values in decreasing order (largest first)

v: Array, unitary / orthogonal

References

N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev., Survey and Review section, Vol. 53, num. 2, pp. 217-288, June 2011 http://arxiv.org/abs/0909.4061

Examples

```
>>> u, s, vt = svd_compressed(x, 20)
```

dask.array.linalg.**tsqr**(*data*, *name=None*, *compute_svd=False*)

Direct Tall-and-Skinny QR algorithm

As presented in:

> A. Benson, D. Gleich, and J. Demmel. Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. IEEE International Conference on Big Data, 2013. http://arxiv.org/abs/1301.1071

This algorithm is used to compute both the QR decomposition and the Singular Value Decomposition. It requires that the input array have a single column of blocks, each of which fit in memory.

Parameters **data: Array**

compute_svd: bool

Whether to compute the SVD rather than the QR decomposition

See also:

dask.array.linalg.qr, *dask.array.linalg.svd*

dask.array.ghost.**ghost**(*x*, *depth*, *boundary*)

Share boundaries between neighboring blocks

Parameters **x: da.Array**

A dask array

depth: dict

The size of the shared boundary per axis

boundary: dict

The boundary condition on each axis. Options are ‘reflect’, ‘periodic’, ‘nearest’, ‘none’, an integer will fill the boundary with that integer.

The axes dict informs how many cells to overlap between neighboring blocks

{0: 2, 2: 5} means share two cells in 0 axis, 5 cells in 2 axis

Examples

```
>>> import numpy as np
>>> import dask.array as da
```

```
>>> x = np.arange(64).reshape((8, 8))
>>> d = da.from_array(x, chunks=(4, 4))
>>> d.chunks
((4, 4), (4, 4))
```

```
>>> g = da.ghost.ghost(d, depth={0: 2, 1: 1},
...                    boundary={0: 100, 1: 'reflect'})
>>> g.chunks
((8, 8), (6, 6))
```

```
>>> np.array(g)
array([[100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [  0,   0,   1,   2,   3,   4,   3,   4,   5,   6,   7,   7],
       [  8,   8,   9,  10,  11,  12,  11,  12,  13,  14,  15,  15],
       [ 16,  16,  17,  18,  19,  20,  19,  20,  21,  22,  23,  23],
       [ 24,  24,  25,  26,  27,  28,  27,  28,  29,  30,  31,  31],
       [ 32,  32,  33,  34,  35,  36,  35,  36,  37,  38,  39,  39],
       [ 40,  40,  41,  42,  43,  44,  43,  44,  45,  46,  47,  47],
       [ 16,  16,  17,  18,  19,  20,  19,  20,  21,  22,  23,  23],
       [ 24,  24,  25,  26,  27,  28,  27,  28,  29,  30,  31,  31],
       [ 32,  32,  33,  34,  35,  36,  35,  36,  37,  38,  39,  39],
       [ 40,  40,  41,  42,  43,  44,  43,  44,  45,  46,  47,  47],
       [ 48,  48,  49,  50,  51,  52,  51,  52,  53,  54,  55,  55],
       [ 56,  56,  57,  58,  59,  60,  59,  60,  61,  62,  63,  63],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]])
```

dask.array.ghost.**map_overlap**(*x*, *func*, *depth*, *boundary=None*, *trim=True*, ***kwargs*)

dask.array.**from_array**(*x*, *chunks*, *name=None*, *lock=False*)

Create dask array from something that looks like an array

Input must have a `.shape` and support numpy-style slicing.

The `chunks` argument must be one of the following forms:

- a blocksize like 1000
- a blockshape like (1000, 1000)
- explicit sizes of all blocks along all dimensions like ((1000, 1000, 500), (400, 400)).

Examples

```
>>> x = h5py.File('...')['/data/path']
>>> a = da.from_array(x, chunks=(1000, 1000))
```

If your underlying datastore does not support concurrent reads then include the `lock=True` keyword argument or `lock=mylock` if you want multiple arrays to coordinate around the same lock.

```
>>> a = da.from_array(x, chunks=(1000, 1000), lock=True)
```

dask.array.**from_delayed**(*value*, *shape*, *dtype=None*, *name=None*)

Create a dask array from a dask delayed value

This routine is useful for constructing dask arrays in an ad-hoc fashion using dask delayed, particularly when combined with stack and concatenate.

The dask array will consist of a single chunk.

Examples

```
>>> from dask import do
>>> value = do(np.ones)(5)
>>> array = from_delayed(value, (5,), dtype=float)
>>> array
dask.array<from-va..., shape=(5,), dtype=float64, chunksize=(5,)>
>>> array.compute()
array([ 1.,  1.,  1.,  1.,  1.])
```

dask.array.**from_npy_stack**(*dirname*, *mmap_mode='r'*)

Load dask array from stack of npy files

See da.to_npy_stack for docstring

Parameters **dirname: string**

Directory of .npy files

mmap_mode: (None or 'r')

Read data in memory map mode

dask.array.**store**(*sources*, *targets*, *lock=True*, *compute=True*, ***kwargs*)

Store dask arrays in array-like objects, overwrite data in target

This stores dask arrays into object that supports numpy-style setitem indexing. It stores values chunk by chunk so that it does not have to fill up memory. For best performance you can align the block size of the storage target with the block size of your array.

If your data fits in memory then you may prefer calling np.array(myarray) instead.

Parameters **sources: Array or iterable of Arrays**

targets: array-like or iterable of array-likes

These should support setitem syntax target[10:20] = ...

lock: boolean or threading.Lock, optional

Whether or not to lock the data stores while storing. Pass True (lock each file individually), False (don't lock) or a particular threading.Lock object to be shared among all writes.

compute: boolean, optional

If true compute immediately, return lazy Value object otherwise

Examples

```
>>> x = ...
```

```
>>> import h5py
>>> f = h5py.File('myfile.hdf5')
>>> dset = f.create_dataset('/data', shape=x.shape,
...                                  chunks=x.chunks,
...                                  dtype='f8')
```

```
>>> store(x, dset)
```

Alternatively store many arrays at the same time

```
>>> store([x, y, z], [dset1, dset2, dset3])
```

dask.array.**to_hdf5**(*filename*, **args*, ***kwargs*)
Store arrays in HDF5 file

This saves several dask arrays into several datapaths in an HDF5 file. It creates the necessary datasets and handles clean file opening/closing.

```
>>> da.to_hdf5('myfile.hdf5', '/x', x)
```

or

```
>>> da.to_hdf5('myfile.hdf5', {'/x': x, '/y': y})
```

Optionally provide arguments as though to `h5py.File.create_dataset`

```
>>> da.to_hdf5('myfile.hdf5', '/x', x, compression='lzf', shuffle=True)
```

This can also be used as a method on a single Array

```
>>> x.to_hdf5('myfile.hdf5', '/x')
```

See also:

`da.store`, `h5py.File.create_dataset`

dask.array.**to_npy_stack**(*dirname*, *x*, *axis=0*)
Write dask array to a stack of .npy files

This partitions the dask.array along one axis and stores each block along that axis as a single .npy file in the specified directory

See also:

from_npy_stack

Examples

```
>>> x = da.ones((5, 10, 10), chunks=(2, 4, 4))
>>> da.to_npy_stack('data/', x, axis=0)
```

```
`bash $ tree data/ data/ |-- 0.npy |-- 1.npy |-- 2.npy |-- info `
```

The `.npy` files store numpy arrays for `x[0:2], x[2:4], and x[4:5]` respectively, as is specified by the chunk size along the zeroth axis. The info file stores the dtype, chunks, and axis information of the array.

You can load these stacks with the `da.from_npy_stack` function.

```
>>> y = da.from_npy_stack('data/')
```

dask.array.fft.**fft**(*a*, *n=None*, *axis=-1*)
Wrapping of numpy.fft.fft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The numpy.fft.fft docstring follows below:

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

Parameters **a** : array_like

Input array, can be complex.

n : int, optional

Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input (along the axis specified by *axis*) is used.

axis : int, optional

Axis over which to compute the FFT. If not given, the last axis is used.

Returns **out** : complex ndarray

The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

Raises **IndexError**

if *axes* is larger than the last axis of *a*.

See also:

numpy.fft for definition of the DFT and conventions used.

ifft The inverse of *fft*.

fft2 The two-dimensional FFT.

fftn The *n*-dimensional FFT.

rfftn The *n*-dimensional FFT of real input.

fftfreq Frequency bins for given FFT parameters.

Notes

FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. The symmetry is highest when *n* is a power of 2, and the transform is therefore most efficient for these sizes.

The DFT is defined, with the conventions used in this implementation, in the documentation for the *numpy.fft* module.

References

[CT]

Examples

```
>>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
array([ -3.44505240e-16 +1.14383329e-17j,
          8.00000000e+00 -5.71092652e-15j,
          2.33482938e-16 +1.22460635e-16j,
          1.64863782e-15 +1.77635684e-15j,
          9.95839695e-17 +2.33482938e-16j,
          0.00000000e+00 +1.66837030e-15j,
```

```
             1.14383329e-17 +1.22460635e-16j,
             -1.64863782e-15 +1.77635684e-15j])
```

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(256)
>>> sp = np.fft.fft(np.sin(t))
>>> freq = np.fft.fftfreq(t.shape[-1])
>>> plt.plot(freq, sp.real, freq, sp.imag)
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>]
>>> plt.show()
```

In this example, real input has an FFT which is Hermitian, i.e., symmetric in the real part and anti-symmetric in the imaginary part, as described in the *numpy.fft* documentation.

`dask.array.fft.`**`ifft`**(*a*, *n=None*, *axis=-1*)

Wrapping of numpy.fft.ifft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The numpy.fft.ifft docstring follows below:

Compute the one-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the one-dimensional *n*-point discrete Fourier transform computed by *fft*. In other words, `ifft(fft(a)) == a` to within numerical accuracy. For a general description of the algorithm and definitions, see *numpy.fft*.

The input should be ordered in the same way as is returned by *fft*, i.e., `a[0]` should contain the zero frequency term, `a[1:n/2+1]` should contain the positive-frequency terms, and `a[n/2+1:]` should contain the negative-frequency terms, in order of decreasingly negative frequency. See *numpy.fft* for details.

Parameters **a** : array_like

Input array, can be complex.

n : int, optional

Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input (along the axis specified by *axis*) is used. See notes about padding issues.

axis : int, optional

Axis over which to compute the inverse DFT. If not given, the last axis is used.

Returns **out** : complex ndarray

The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

Raises **IndexError**

If *axes* is larger than the last axis of *a*.

See also:

`numpy.fft` An introduction, with definitions and general explanations.

`fft` The one-dimensional (forward) FFT, of which *ifft* is the inverse

`ifft2` The two-dimensional inverse FFT.

ifftn The n-dimensional inverse FFT.

Notes

If the input parameter *n* is larger than the size of the input, the input is padded by appending zeros at the end. Even though this is the common approach, it might lead to surprising results. If a different padding is desired, it must be performed before calling *ifft*.

Examples

```
>>> np.fft.ifft([0, 4, 0, 0])
array([ 1.+0.j,  0.+1.j, -1.+0.j,  0.-1.j])
```

Create and plot a band-limited signal with random phases:

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(400)
>>> n = np.zeros((400,), dtype=complex)
>>> n[40:60] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20,)))
>>> s = np.fft.ifft(n)
>>> plt.plot(t, s.real, 'b-', t, s.imag, 'r--')
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>]
>>> plt.legend(('real', 'imaginary'))
<matplotlib.legend.Legend object at 0x...>
>>> plt.show()
```

dask.array.fft.**hfft**(*a*, *n=None*, *axis=-1*)

Wrapping of numpy.fft.hfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The numpy.fft.hfft docstring follows below:

Compute the FFT of a signal whose spectrum has Hermitian symmetry.

Parameters **a** : array_like

The input array.

n : int, optional

The length of the FFT.

axis : int, optional

The axis over which to compute the FFT, assuming Hermitian symmetry of the spectrum. Default is the last axis.

Returns **out** : ndarray

The transformed input.

See also:

rfft Compute the one-dimensional FFT for real input.

ihfft The inverse of *hfft*.

Notes

hfft/*ihfft* are a pair analogous to *rfft*/*irfft*, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it's *hfft* for which you must supply the length of the result if it is to be odd: `ihfft(hfft(a), len(a)) == a`, within numerical accuracy.

Examples

```
>>> signal = np.array([[1, 1.j], [-1.j, 2]])
>>> np.conj(signal.T) - signal   # check Hermitian symmetry
array([[ 0.-0.j,  0.+0.j],
       [ 0.+0.j,  0.-0.j]])
>>> freq_spectrum = np.fft.hfft(signal)
>>> freq_spectrum
array([[ 1.,  1.],
       [ 2., -2.]])
```

dask.array.fft.**ihfft**(*a*, *n=None*, *axis=-1*)

Wrapping of numpy.fft.ihfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The numpy.fft.ihfft docstring follows below:

Compute the inverse FFT of a signal whose spectrum has Hermitian symmetry.

Parameters **a** : array_like

Input array.

n : int, optional

Length of the inverse FFT.

axis : int, optional

Axis over which to compute the inverse FFT, assuming Hermitian symmetry of the spectrum. Default is the last axis.

Returns **out** : ndarray

The transformed input.

See also:

hfft, *irfft*

Notes

hfft/*ihfft* are a pair analogous to *rfft*/*irfft*, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it's *hfft* for which you must supply the length of the result if it is to be odd: `ihfft(hfft(a), len(a)) == a`, within numerical accuracy.

dask.array.fft.**rfft**(*a*, *n=None*, *axis=-1*)

Wrapping of numpy.fft.rfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The numpy.fft.rfft docstring follows below:

Compute the one-dimensional discrete Fourier Transform for real input.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) of a real-valued array by means of an efficient algorithm called the Fast Fourier Transform (FFT).

Parameters **a** : array_like

Input array

n : int, optional

Number of points along transformation axis in the input to use. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input (along the axis specified by *axis*) is used.

axis : int, optional

Axis over which to compute the FFT. If not given, the last axis is used.

Returns **out** : complex ndarray

The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. If *n* is even, the length of the transformed axis is `(n/2)+1`. If *n* is odd, the length is `(n+1)/2`.

Raises **IndexError**

If *axis* is larger than the last axis of *a*.

See also:

`numpy.fft` For definition of the DFT and conventions used.

`irfft` The inverse of *rfft*.

`fft` The one-dimensional FFT of general (complex) input.

`fftn` The *n*-dimensional FFT.

`rfftn` The *n*-dimensional FFT of real input.

Notes

When the DFT is computed for purely real input, the output is Hermite-symmetric, i.e. the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency terms are therefore redundant. This function does not compute the negative frequency terms, and the length of the transformed axis of the output is therefore `n//2+1`.

When `A = rfft(a)` and fs is the sampling frequency, `A[0]` contains the zero-frequency term 0*fs, which is real due to Hermitian symmetry.

If *n* is even, `A[-1]` contains the term representing both positive and negative Nyquist frequency (+fs/2 and -fs/2), and must also be purely real. If *n* is odd, there is no term at fs/2; `A[-1]` contains the largest positive frequency (fs/2*(n-1)/n), and is complex in the general case.

If the input *a* contains an imaginary part, it is silently discarded.

Examples

```
>>> np.fft.fft([0, 1, 0, 0])
array([ 1.+0.j,  0.-1.j, -1.+0.j,  0.+1.j])
>>> np.fft.rfft([0, 1, 0, 0])
array([ 1.+0.j,  0.-1.j, -1.+0.j])
```

Notice how the final element of the *fft* output is the complex conjugate of the second element, for real input. For *rfft*, this symmetry is exploited to compute only the non-negative frequency terms.

`dask.array.fft.`**`irfft`**(*a*, *n=None*, *axis=-1*)

Wrapping of numpy.fft.irfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The numpy.fft.irfft docstring follows below:

Compute the inverse of the n-point DFT for real input.

This function computes the inverse of the one-dimensional *n*-point discrete Fourier Transform of real input computed by *rfft*. In other words, `irfft(rfft(a), len(a)) == a` to within numerical accuracy. (See Notes below for why `len(a)` is necessary here.)

The input is expected to be in the form returned by *rfft*, i.e. the real zero-frequency term followed by the complex positive frequency terms in order of increasing frequency. Since the discrete Fourier Transform of real input is Hermite-symmetric, the negative frequency terms are taken to be the complex conjugates of the corresponding positive frequency terms.

Parameters **a** : array_like

The input array.

n : int, optional

Length of the transformed axis of the output. For *n* output points, `n//2+1` input points are necessary. If the input is longer than this, it is cropped. If it is shorter than this, it is padded with zeros. If *n* is not given, it is determined from the length of the input (along the axis specified by *axis*).

axis : int, optional

Axis over which to compute the inverse FFT.

Returns **out** : ndarray

The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. The length of the transformed axis is *n*, or, if *n* is not given, `2*(m-1)` where *m* is the length of the transformed axis of the input. To get an odd number of output points, *n* must be specified.

Raises **IndexError**

If *axis* is larger than the last axis of *a*.

See also:

`numpy.fft` For definition of the DFT and conventions used.

`rfft` The one-dimensional FFT of real input, of which *irfft* is inverse.

`fft` The one-dimensional FFT.

`irfft2` The inverse of the two-dimensional FFT of real input.

irfftn The inverse of the *n*-dimensional FFT of real input.

Notes

Returns the real valued *n*-point inverse discrete Fourier transform of *a*, where *a* contains the non-negative frequency terms of a Hermite-symmetric sequence. *n* is the length of the result, not the input.

If you specify an *n* such that *a* must be zero-padded or truncated, the extra/removed values will be added/removed at high frequencies. One can thus resample a series to *m* points via Fourier interpolation by: `a_resamp = irfft(rfft(a), m)`.

Examples

```
>>> np.fft.ifft([1, -1j, -1, 1j])
array([ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j])
>>> np.fft.irfft([1, -1j, -1])
array([ 0.,  1.,  0.,  0.])
```

Notice how the last term in the input to the ordinary *ifft* is the complex conjugate of the second term, and the output has zero imaginary part everywhere. When calling *irfft*, the negative frequencies are not specified, and the output array is purely real.

dask.array.random.**beta**(*a*, *b*, *size=None*)
The Beta distribution over `[0, 1]`.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1} dt.$$

It is often seen in Bayesian inference and order statistics.

Parameters **a** : float

Alpha, non-negative.

b : float

Beta, non-negative.

size : tuple of ints, optional

The number of samples to draw. The output is packed according to the size given.

Returns **out** : ndarray

Array of the given shape, containing values drawn from a Beta distribution.

dask.array.random.**binomial**(*n*, *p*, *size=None*)
Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

Parameters **n** : float (but truncated to an integer)

parameter, >= 0.

p : float

parameter, >= 0 and <=1.

size : {tuple, int}

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : {ndarray, scalar}

where the values are all integers in [0, n].

See also:

`scipy.stats.distributions.binom` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

References

[R95], *[R96]*, *[R97]*, *[R98]*, *[R99]*

Examples

Draw samples from the distribution:

>> n, p = 10, .5 # number of trials, probability of each trial >> s = np.random.binomial(n, p, 1000) # result of flipping a coin 10 times, tested 1000 times.

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

>> sum(np.random.binomial(9,0.1,20000)==0)/20000. answer = 0.38885, or 38%.

`dask.array.random.`**`chisquare`**(*df*, *size=None*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Parameters **df** : int

Number of degrees of freedom.

size : tuple of ints, int, optional

Size of the returned array. By default, a scalar is returned.

Returns **output** : ndarray

Samples drawn from the distribution, packed in a *size*-shaped array.

Raises **ValueError**

When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\texttt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

References

NIST/SEMATECH e-Handbook of Statistical Methods

Examples

>> np.random.chisquare(2,4) array([1.89920014, 9.00867716, 3.13710533, 5.62318272])

`dask.array.random.`**`exponential`**(*scale=1.0*, *size=None*)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution *[R102]*.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms *[R100]*, or the time between page requests to Wikipedia *[R101]*.

Parameters **scale** : float

The scale parameter, $\beta = 1/\lambda$.

size : tuple of ints

Number of samples to draw. The output is shaped according to *size*.

References

[R100], *[R101]*, *[R102]*

`dask.array.random.`**`f`**(*dfnum*, *dfden*, *size=None*)

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

Parameters **dfnum** : float

Degrees of freedom in numerator. Should be greater than zero.

dfden : float

Degrees of freedom in denominator. Should be greater than zero.

size : {tuple, int}, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. By default only one sample is returned.

Returns **samples** : {ndarray, scalar}

Samples from the Fisher distribution.

See also:

`scipy.stats.distributions.f` probability density function, distribution or cumulative density function, etc.

Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

References

[R103], *[R104]*

Examples

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

>> dfnum = 1. # between group degrees of freedom >> dfden = 48. # within groups degrees of freedom >> s = np.random.f(dfnum, dfden, 1000)

The lower bound for the top 1% of the samples is :

>> sort(s)[-10] 7.61988120985

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

`dask.array.random.`**`gamma`**(*shape*, *scale=1.0*, *size=None*)

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

Parameters **shape** : scalar > 0

The shape of the gamma distribution.

scale : scalar > 0, optional

The scale of the gamma distribution. Default is equal to 1.

size : shape_tuple, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **out** : ndarray, float

Returns one sample unless *size* parameter is specified.

See also:

`scipy.stats.distributions.gamma` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

[R105], *[R106]*

Examples

Draw samples from the distribution:

>> shape, scale = 2., 2. # mean and dispersion >> s = np.random.gamma(shape, scale, 1000)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> import scipy.special as sps >> count, bins, ignored = plt.hist(s, 50, normed=True) >> y = bins**(shape-1)*(np.exp(-bins/scale) / .. (sps.gamma(shape)*scale**shape)) >> plt.plot(bins, y, linewidth=2, color='r') >> plt.show()

`dask.array.random.`**`geometric`**(*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where *p* is the probability of success of an individual trial.

Parameters **p** : float

The probability of success of an individual trial.

size : tuple of ints

Number of values to draw from the distribution. The output is shaped according to *size*.

Returns **out** : ndarray

Samples from the geometric distribution, shaped according to *size*.

Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

>> z = np.random.geometric(p=0.35, size=10000)

How many trials succeeded after a single run?

>> (z == 1).sum() / 10000. 0.34889999999999999 #random

`dask.array.random.`**`gumbel`**(*loc=0.0*, *scale=1.0*, *size=None*)

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

Parameters **loc** : float

The location of the mode of the distribution.

scale : float

The scale parameter of the distribution.

size : tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **out** : ndarray

The samples

See also:

`scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`

`scipy.stats.genextreme` probability density function, distribution, or cumulative density function, etc. for each of the above

`weibull`

Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

References

Gumbel, E. J., *Statistics of Extremes*, New York: Columbia University Press, 1958.

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields*, Basel: Birkhauser Verlag, 2001.

Examples

Draw samples from the distribution:

>> mu, beta = 0, 0.1 # location and scale >> s = np.random.gumbel(mu, beta, 1000)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> count, bins, ignored = plt.hist(s, 30, normed=True) >> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta) .. * np.exp(-np.exp(-(bins - mu) /beta)), .. linewidth=2, color='r') >> plt.show()

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

>> means = [] >> maxima = [] >> for i in range(0,1000) : .. a = np.random.normal(mu, beta, 1000) .. means.append(a.mean()) .. maxima.append(a.max()) >> count, bins, ignored = plt.hist(maxima, 30, normed=True) >> beta = np.std(maxima)*np.pi/np.sqrt(6) >> mu = np.mean(maxima) - 0.57721*beta >> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta) .. * np.exp(-np.exp(-(bins - mu)/beta)), .. linewidth=2, color='r') >> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi)) .. * np.exp(-(bins - mu)**2 / (2 * beta**2)), .. linewidth=2, color='g') >> plt.show()

`dask.array.random.`**`hypergeometric`**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

Parameters **ngood** : int or array_like

Number of ways to make a good selection. Must be nonnegative.

nbad : int or array_like

Number of ways to make a bad selection. Must be nonnegative.

nsample : int or array_like

Number of items sampled. Must be at least 1 and at most `ngood + nbad`.

size : int or tuple of int

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : ndarray or scalar

The values are all integers in [0, n].

See also:

`scipy.stats.distributions.hypergeom` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n}\binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \leq x \leq m$ and $n + m - N \leq x \leq n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

References

[R107], *[R108]*, *[R109]*

Examples

Draw samples from the distribution:

>> ngood, nbad, nsamp = 100, 2, 10 # number of good, number of bad, and number of samples >> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000) >> hist(s) # note that it is very unlikely to grab both bad items

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

>> s = np.random.hypergeometric(15, 15, 15, 100000) >> sum(s>=12)/100000. + sum(s<=3)/100000. # answer = 0.003 .. pretty unlikely!

`dask.array.random.`**`laplace`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

Parameters **loc** : float

The position, μ, of the distribution peak.

scale : float

λ, the exponential decay.

Notes

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution

References

[R110], *[R111]*, *[R112]*, *[R113]*

Examples

Draw samples from the distribution

>> loc, scale = 0., 1. >> s = np.random.laplace(loc, scale, 1000)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> count, bins, ignored = plt.hist(s, 30, normed=True) >> x = np.arange(-8., 8., .01) >> pdf = np.exp(-abs(x-loc/scale))/(2.*scale) >> plt.plot(x, pdf)

Plot Gaussian for comparison:

>> g = (1/(scale * np.sqrt(2 * np.pi)) * .. np.exp(- (x - loc)**2 / (2 * scale**2))) >> plt.plot(x,g)

`dask.array.random.`**`logistic`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Parameters **loc** : float

scale : float > 0.

size : {tuple, int}

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : {ndarray, scalar}

where the values are all integers in [0, n].

See also:

`scipy.stats.distributions.logistic` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

References

[R114], *[R115]*, *[R116]*

Examples

Draw samples from the distribution:

>> loc, scale = 10, 1 >> s = np.random.logistic(loc, scale, 10000) >> count, bins, ignored = plt.hist(s, bins=50)

plot against distribution

>> def logist(x, loc, scale): .. return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2) >> plt.plot(bins, logist(bins, loc, scale)*count.max()/.. logist(bins, loc, scale).max()) >> plt.show()

`dask.array.random.`**`lognormal`**(*mean=0.0*, *sigma=1.0*, *size=None*)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

Parameters **mean** : float

Mean value of the underlying normal distribution

sigma : float, > 0.

Standard deviation of the underlying normal distribution

size : tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : ndarray or float

The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

See also:

`scipy.stats.lognorm` probability density function, distribution, cumulative density function, etc.

Notes

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x\sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

References

Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

Reiss, R.D. and Thomas, M., *Statistical Analysis of Extreme Values*, Basel: Birkhauser Verlag, 2001, pp. 31-32.

Examples

Draw samples from the distribution:

>> mu, sigma = 3., 1. # mean and standard deviation >> s = np.random.lognormal(mu, sigma, 1000)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')

>> x = np.linspace(min(bins), max(bins), 10000) >> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2)) .. / (x * sigma * np.sqrt(2 * np.pi)))

>> plt.plot(x, pdf, linewidth=2, color='r') >> plt.axis('tight') >> plt.show()

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

>> # Generate a thousand samples: each is the product of 100 random >> # values, drawn from a normal distribution. >> b = [] >> for i in range(1000): .. a = 10. + np.random.random(100) .. b.append(np.product(a))

>> b = np.array(b) / np.min(b) # scale values to be positive >> count, bins, ignored = plt.hist(b, 100, normed=True, align='center') >> sigma = np.std(np.log(b)) >> mu = np.mean(np.log(b))

>> x = np.linspace(min(bins), max(bins), 10000) >> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2)) .. / (x * sigma * np.sqrt(2 * np.pi)))

>> plt.plot(x, pdf, color='r', linewidth=2) >> plt.show()

`dask.array.random.`**`logseries`**(*p*, *size=None*)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0 < p < 1).

Parameters **loc** : float

scale : float > 0.

size : {tuple, int}

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : {ndarray, scalar}

where the values are all integers in [0, n].

See also:

`scipy.stats.distributions.logser` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

References

[R117], *[R118]*, *[R119]*, *[R120]*

Examples

Draw samples from the distribution:

>> a = .6 >> s = np.random.logseries(a, 10000) >> count, bins, ignored = plt.hist(s)

plot against distribution

>> def logseries(k, p): .. return -p**k/(k*log(1-p)) >> plt.plot(bins, logseries(bins, a)*count.max()/

logseries(bins, a).max(), 'r')

>> plt.show()

`dask.array.random.`**`negative_binomial`**(*n*, *p*, *size=None*)

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, *n* trials and *p* probability of success where *n* is an integer > 0 and *p* is in the interval [0, 1].

Parameters **n** : int

Parameter, > 0.

p : float

Parameter, >= 0 and <=1.

size : int or tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : int or ndarray of ints

Drawn samples.

Notes

The probability density for the Negative Binomial distribution is

$$P(N;n,p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where $n-1$ is the number of successes, p is the probability of success, and $N+n-1$ is the number of trials.

The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

References

[R121], *[R122]*

Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

>> s = np.random.negative_binomial(1, 0.1, 100000) >> for i in range(1, 11): .. probability = sum(s<i) / 100000. .. print i, "wells drilled, probability of one success =", probability

`dask.array.random.`**`noncentral_chisquare`**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

Parameters **df** : int

Degrees of freedom, should be >= 1.

nonc : float

Non-centrality, should be > 0.

size : int or tuple of ints

Shape of the output.

Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

References

[R123], *[R124]*

Examples

Draw values from the distribution and plot the histogram

>> import matplotlib.pyplot as plt >> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000), .. bins=200, normed=True) >> plt.show()

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

>> plt.figure() >> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000), .. bins=np.arange(0., 25, .1), normed=True) >> values2 = plt.hist(np.random.chisquare(3, 100000), .. bins=np.arange(0., 25, .1), normed=True) >> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob') >> plt.show()

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

>> plt.figure() >> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000), .. bins=200, normed=True) >> plt.show()

dask.array.random.**noncentral_f**(*dfnum*, *dfden*, *nonc*, *size=None*)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

Parameters **dfnum** : int

Parameter, should be > 1.

dfden : int

Parameter, should be > 1.

nonc : float

Parameter, should be >= 0.

size : int or tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : scalar or ndarray

Drawn samples.

Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

References

Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

>> dfnum = 3 # between group deg of freedom >> dfden = 20 # within groups degrees of freedom >> nonc = 3.0 >> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000) >> NF = np.histogram(nc_vals, bins=50, normed=True) >> c_vals = np.random.f(dfnum, dfden, 1000000) >> F = np.histogram(c_vals, bins=50, normed=True) >> plt.plot(F[1][1:], F[0]) >> plt.plot(NF[1][1:], NF[0]) >> plt.show()

dask.array.random.**normal**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently *[R126]*, is often called the bell curve because of its characteristic shape (see the example below).

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 *[R126]*.

Parameters **loc** : float

Mean ("centre") of the distribution.

scale : float

Standard deviation (spread or "width") of the distribution.

size : tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

See also:

`scipy.stats.distributions.norm` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ *[R126]*). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

References

[R125], *[R126]*

Examples

Draw samples from the distribution:

>> mu, sigma = 0, 0.1 # mean and standard deviation >> s = np.random.normal(mu, sigma, 1000)

Verify the mean and the variance:

>> abs(mu - np.mean(s)) < 0.01 True

>> abs(sigma - np.std(s, 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(s, 30, normed=True) >> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) * .. np.exp(- (bins - mu)**2 / (2 * sigma**2)), .. linewidth=2, color='r') >> plt.show()

dask.array.random.**pareto**(*a*, *size=None*)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter m, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is m, where the standard Pareto distribution has location m=1. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

Parameters **shape** : float, > 0.

Shape of the distribution.

size : tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

See also:

scipy.stats.distributions.lomax.pdf probability density function, distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

References

[R127], *[R128]*, *[R129]*, *[R130]*

Examples

Draw samples from the distribution:

>> a, m = 3., 1. # shape and mode >> s = np.random.pareto(a, 1000) + m

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> count, bins, ignored = plt.hist(s, 100, normed=True, align='center') >> fit = a*m**a/bins**(a+1) >> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r') >> plt.show()

dask.array.random.**poisson**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

Parameters **lam** : float

Expectation of interval, should be >= 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Notes

The Poisson distribution

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ.

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

References

[R131], *[R132]*

Examples

Draw samples from the distribution:

>> import numpy as np >> s = np.random.poisson(5, 10000)

Display histogram of the sample:

>> import matplotlib.pyplot as plt >> count, bins, ignored = plt.hist(s, 14, normed=True) >> plt.show()

dask.array.random.**power**(*a*, *size=None*)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

Parameters **a** : float

parameter, > 0

size : tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : {ndarray, scalar}

The returned samples lie in [0, 1].

Raises **ValueError**

If a<1.

Notes

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \leq x \leq 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

References

[R133], *[R134]*

Examples

Draw samples from the distribution:

>> a = 5. # shape >> samples = 1000 >> s = np.random.power(a, samples)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> count, bins, ignored = plt.hist(s, bins=30) >> x = np.linspace(0, 1, 100) >> y = a*x**(a-1.) >> normed_y = samples*np.diff(bins)[0]*y >> plt.plot(x, normed_y) >> plt.show()

Compare the power function distribution to the inverse of the Pareto.

>> from scipy import stats >> rvs = np.random.power(5, 1000000) >> rvsp = np.random.pareto(5, 1000000) >> xx = np.linspace(0,1,100) >> powpdf = stats.powerlaw.pdf(xx,5)

>> plt.figure() >> plt.hist(rvs, bins=50, normed=True) >> plt.plot(xx,powpdf,'r-') >> plt.title('np.random.power(5)')

>> plt.figure() >> plt.hist(1./(1.+rvsp), bins=50, normed=True) >> plt.plot(xx,powpdf,'r-') >> plt.title('inverse of 1 + np.random.pareto(5)')

>> plt.figure() >> plt.hist(1./(1.+rvsp), bins=50, normed=True) >> plt.plot(xx,powpdf,'r-') >> plt.title('inverse of stats.pareto(5)')

`dask.array.random.`**`product`**`()`

product(*iterables) –> product object

Cartesian product of input iterables. Equivalent to nested for-loops.

For example, product(A, B) returns the same as: ((x,y) for x in A for y in B). The leftmost iterators are in the outermost for-loop, so the output tuples cycle in a manner similar to an odometer (with the rightmost element changing on every iteration).

To compute the product of an iterable with itself, specify the number of repetitions with the optional repeat keyword argument. For example, product(A, repeat=4) means the same as product(A, A, A, A).

product('ab', range(3)) –> ('a',0) ('a',1) ('a',2) ('b',0) ('b',1) ('b',2) product((0,1), (0,1), (0,1)) –> (0,0,0) (0,0,1) (0,1,0) (0,1,1) (1,0,0) ...

dask.array.random.**random**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Parameters **size** : int or tuple of ints, optional

Defines the shape of the returned array of random floats. If None (the default), returns a single float.

Returns **out** : float or ndarray of floats

Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Examples

>> np.random.random_sample() 0.47108547995356098 >> type(np.random.random_sample()) <type 'float'> >> np.random.random_sample((5,)) array([0.30220482, 0.86820401, 0.1654503 , 0.11659149, 0.54323428])

Three-by-two array of random numbers from [-5, 0):

>> 5 * np.random.random_sample((3, 2)) - 5 array([[-3.99149989, -0.52338984],

[-2.99091858, -0.79479508], [-1.23204345, -1.75224494]])

dask.array.random.**random_sample**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Parameters **size** : int or tuple of ints, optional

Defines the shape of the returned array of random floats. If None (the default), returns a single float.

Returns **out** : float or ndarray of floats

Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Examples

>> np.random.random_sample() 0.47108547995356098 >> type(np.random.random_sample()) <type 'float'> >> np.random.random_sample((5,)) array([0.30220482, 0.86820401, 0.1654503 , 0.11659149, 0.54323428])

Three-by-two array of random numbers from [-5, 0):

>> 5 * np.random.random_sample((3, 2)) - 5 array([[-3.99149989, -0.52338984],

[-2.99091858, -0.79479508], [-1.23204345, -1.75224494]])

dask.array.random.**rayleigh**(*scale=1.0*, *size=None*)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

Parameters **scale** : scalar

Scale, also equals the mode. Should be >= 0.

size : int or tuple of ints, optional

Shape of the output. Default is None, in which case a single value is returned.

Notes

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

References

[R135], *[R136]*

Examples

Draw values from the distribution and plot the histogram

>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

>> meanvalue = 1 >> modevalue = np.sqrt(2 / np.pi) * meanvalue >> s = np.random.rayleigh(modevalue, 1000000)

The percentage of waves larger than 3 meters is:

>> 100.*sum(s>3)/1000000. 0.087300000000000003

dask.array.random.**standard_cauchy**(*size=None*)

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

Parameters **size** : int or tuple of ints

Shape of the output.

Returns **samples** : ndarray or scalar

The drawn samples.

Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + (\frac{x-x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

References

[R137], *[R138]*, *[R139]*

Examples

Draw samples and plot the distribution:

>> s = np.random.standard_cauchy(1000000) >> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well >> plt.hist(s, bins=100) >> plt.show()

`dask.array.random.`**`standard_exponential`**(*size=None*)
Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Parameters **size** : int or tuple of ints

Shape of the output.

Returns **out** : float or ndarray

Drawn samples.

Examples

Output a 3x8000 array:

>> n = np.random.standard_exponential((3, 8000))

`dask.array.random.`**`standard_gamma`**(*shape*, *size=None*)
Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

Parameters **shape** : float

Parameter, should be > 0.

size : int or tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : ndarray or scalar

The drawn samples.

See also:

`scipy.stats.distributions.gamma` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

[R140], *[R141]*

Examples

Draw samples from the distribution:

>> shape, scale = 2., 1. # mean and width >> s = np.random.standard_gamma(shape, 1000000)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> import scipy.special as sps >> count, bins, ignored = plt.hist(s, 50, normed=True) >> y = bins**(shape-1) * ((np.exp(-bins/scale))/ .. (sps.gamma(shape) * scale**shape)) >> plt.plot(bins, y, linewidth=2, color='r') >> plt.show()

`dask.array.random.`**`standard_normal`**(*size=None*)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

Parameters **size** : int or tuple of ints, optional

Output shape. Default is None, in which case a single value is returned.

Returns **out** : float or ndarray

Drawn samples.

Examples

>> s = np.random.standard_normal(8000) >> s array([0.6888893 , 0.78096262, -0.89086505, .., 0.49876311, #random

-0.38672696, -0.4685006]) #random

>> s.shape (8000,) >> s = np.random.standard_normal(size=(3, 4, 2)) >> s.shape (3, 4, 2)

dask.array.random.**standard_t**(*df*, *size=None*)

Standard Student's t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

Parameters **df** : int

Degrees of freedom, should be > 0.

size : int or tuple of ints, optional

Output shape. Default is None, in which case a single value is returned.

Returns **samples** : ndarray or scalar

Drawn samples.

Notes

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\,\Gamma(\frac{df}{2})}\left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

References

[R142], *[R143]*

Examples

From Dalgaard page 83 *[R142]*, suppose the daily energy intake for 11 women in Kj is:

>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, .. 7515, 8230, 8770])

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

>> s = np.random.standard_t(10, size=100000) >> np.mean(intake) 6753.636363636364 >> intake.std(ddof=1) 1142.1232221373727

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake))) >> import matplotlib.pyplot as plt >> h = plt.hist(s, bins=100, normed=True)

For a one-sided t-test, how far out in the distribution does the t statistic appear?

>> >> np.sum(s<t) / float(len(s)) 0.0090699999999999999 #random

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

dask.array.random.**triangular**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

Parameters **left** : scalar

Lower limit.

mode : scalar

The value where the peak of the distribution occurs. The value should fulfill the condition `left <= mode <= right`.

right : scalar

Upper limit, should be larger than *left*.

size : int or tuple of ints, optional

Output shape. Default is None, in which case a single value is returned.

Returns **samples** : ndarray or scalar

The returned samples all lie in the interval [left, right].

Notes

The probability density function for the Triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

References

[R144]

Examples

Draw values from the distribution and plot the histogram:

>> import matplotlib.pyplot as plt >> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200, .. normed=True) >> plt.show()

dask.array.random.**uniform**(*low=0.0*, *high=1.0*, *size=1*)

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, optional

Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high : float

Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size : int or tuple of ints, optional

Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

Returns **out** : ndarray

Drawn samples, with shape *size*.

See also:

randint Discrete uniform distribution, yielding integers.

random_integers Discrete uniform distribution over the closed interval `[low, high]`.

random_sample Floats uniformly distributed over `[0, 1)`.

random Alias for *random_sample*.

rand Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

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.

Examples

Draw samples from the distribution:

>> s = np.random.uniform(-1,0,1000)

All values are within the given interval:

>> np.all(s >= -1) True >> np.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()

`dask.array.random.`**vonmises**(*mu*, *kappa*, *size=None*)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

Parameters **mu** : float

Mode ("center") of the distribution.

kappa : float

Dispersion of the distribution, has to be >=0.

size : int or tuple of int

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns **samples** : scalar or ndarray

The returned samples, which are in the interval [-pi, pi].

See also:

`scipy.stats.distributions.vonmises` probability density function, distribution, or cumulative density function, etc.

Notes

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

References

Abramowitz, M. and Stegun, I. A. (ed.), *Handbook of Mathematical Functions*, New York: Dover, 1965.

von Mises, R., *Mathematical Theory of Probability and Statistics*, New York: Academic Press, 1964.

Examples

Draw samples from the distribution:

>> mu, kappa = 0.0, 4.0 # mean and dispersion >> s = np.random.vonmises(mu, kappa, 1000)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> import scipy.special as sps >> count, bins, ignored = plt.hist(s, 50, normed=True) >> x = np.arange(-np.pi, np.pi, 2*np.pi/50.) >> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa)) >> plt.plot(x, y/max(y), linewidth=2, color='r') >> plt.show()

`dask.array.random.`**`wald`**(*mean*, *scale*, *size=None*)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Parameters **mean** : scalar

Distribution mean, should be > 0.

scale : scalar

Scale parameter, should be >= 0.

size : int or tuple of ints, optional

Output shape. Default is None, in which case a single value is returned.

Returns **samples** : ndarray or scalar

Drawn sample, all greater than zero.

Notes

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

References

[R145], *[R146]*, *[R147]*

Examples

Draw values from the distribution and plot the histogram:

>> import matplotlib.pyplot as plt >> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True) >> plt.show()

`dask.array.random.`**`weibull`**(*a*, *size=None*)

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

Parameters **a** : float

Shape of the distribution.

size : tuple of ints

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

See also:

`scipy.stats.distributions.weibull_max`, `scipy.stats.distributions.weibull_min`, `scipy.stats.distributions.genextreme`, *gumbel*

Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

References

[R148], *[R149]*, *[R150]*

Examples

Draw samples from the distribution:

>> a = 5. # shape >> s = np.random.weibull(a, 1000)

Display the histogram of the samples, along with the probability density function:

>> import matplotlib.pyplot as plt >> x = np.arange(1,100.)/50. >> def weib(x,n,a): .. return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)

>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000)) >> x = np.arange(1,100.)/50. >> scale = count.max()/weib(x, 1., 5.).max() >> plt.plot(x, weib(x, 1., 5.)*scale) >> plt.show()

`dask.array.random.`**`zipf`**(*a*, *size=None*)
Standard distributions

`dask.array.core.`**`map_blocks`**(*func*, **args*, ***kwargs*)
Map a function across all blocks of a dask array

Parameters **func: callable**

Function to apply to every block in the array

args: dask arrays or constants

dtype: np.dtype

Datatype of resulting array

chunks: tuple (optional)

chunk shape of resulting blocks if the function does not preserve shape

drop_axis: number or iterable (optional)

Dimensions lost by the function

new_axis: number or iterable (optional)

New dimensions created by the function

****kwargs:**

Other keyword arguments to pass to function. Values must be constants (not dask.arrays)

You must also specify the chunks and dtype of the resulting array. If you don't then we assume that the resulting array has the same block structure as the input.

Examples

```
>>> import dask.array as da
>>> x = da.arange(6, chunks=3)
```

```
>>> x.map_blocks(lambda x: x * 2).compute()
array([ 0,  2,  4,  6,  8, 10])
```

The `da.map_blocks` function can also accept multiple arrays

```
>>> d = da.arange(5, chunks=2)
>>> e = da.arange(5, chunks=2)
```

```
>>> f = map_blocks(lambda a, b: a + b**2, d, e)
>>> f.compute()
array([ 0,  2,  6, 12, 20])
```

If function changes shape of the blocks then please provide chunks explicitly.

```
>>> y = x.map_blocks(lambda x: x[::2], chunks=((2, 2),))
```

You have a bit of freedom in specifying chunks. If all of the output chunk sizes are the same, you can provide just that chunk size as a single tuple.

```
>>> a = da.arange(18, chunks=(6,))
>>> b = a.map_blocks(lambda x: x[:3], chunks=(3,))
```

If the function changes the dimension of the blocks you must specify the created or destroyed dimensions.

```
>>> b = a.map_blocks(lambda x: x[None, :, None], chunks=(1, 6, 1),
...                  new_axis=[0, 2])
```

Map_blocks aligns blocks by block positions without regard to shape. In the following example we have two arrays with the same number of blocks but with different shape and chunk sizes.

```
>>> x = da.arange(1000, chunks=(100,))
>>> y = da.arange(100, chunks=(10,))
```

The relevant attribute to match is numblocks

```
>>> x.numblocks
(10,)
>>> y.numblocks
(10,)
```

If these must match (up to broadcasting rules) then we can map arbitrary functions across blocks

```
>>> def func(a, b):
...     return np.array([a.max(), b.max()])
```

```
>>> da.map_blocks(func, x, y, chunks=(2,), dtype='i8')
dask.array<..., shape=(20,), dtype=int64, chunksize=(2,)>
```

```
>>> _.compute()
array([ 99,   9, 199,  19, 299,  29, 399,  39, 499,  49, 599,  59, 699,
        69, 799,  79, 899,  89, 999,  99])
```

Your block function can learn where in the array it is if it supports a `block_id` keyword argument. This will receive entries like (2, 0, 1), the position of the block in the dask array.

```
>>> def func(block, block_id=None):
...     pass
```

You may specify the name of the resulting task in the graph with the optional `name` keyword argument.

```
>>> y = x.map_blocks(lambda x: x + 1, name='increment')
```

`dask.array.core.`**`atop`**(*func*, *out_ind*, **args*, ***kwargs*)

Tensor operation: Generalized inner and outer products

A broad class of blocked algorithms and patterns can be specified with a concise multi-index notation. The `atop` function applies an in-memory function across multiple blocks of multiple inputs in a variety of ways.

Parameters **func: callable**

Function to apply to individual tuples of blocks

out_ind: iterable

Block pattern of the output, something like 'ijk' or (1, 2, 3)

***args: sequence of Array, index pairs**

Sequence like (x, 'ij', y, 'jk', z, 'i')

****kwargs: dict**

Extra keyword arguments to pass to function

This is best explained through example. Consider the following examples:

See also:

top, `contains`

Examples

2D embarrassingly parallel operation from two arrays, x, and y.

```
>>> z = atop(operator.add, 'ij', x, 'ij', y, 'ij')  # z = x + y
```

Outer product multiplying x by y, two 1-d vectors

```
>>> z = atop(operator.mul, 'ij', x, 'i', y, 'j')
```

z = x.T

```
>>> z = atop(np.transpose, 'ji', x, 'ij')
```

The transpose case above is illustrative because it does same transposition both on each in-memory block by calling `np.transpose` and on the order of the blocks themselves, by switching the order of the index `ij -> ji`.

We can compose these same patterns with more variables and more complex in-memory functions

z = X + Y.T

```
>>> z = atop(lambda x, y: x + y.T, 'ij', x, 'ij', y, 'ji')
```

Any index, like `i` missing from the output index is interpreted as a contraction (note that this differs from Einstein convention; repeated indices do not imply contraction.) In the case of a contraction the passed function should expect an iterator of blocks on any array that holds that index.

Inner product multiplying x by y, two 1-d vectors

```
>>> def sequence_dot(x_blocks, y_blocks):
...     result = 0
...     for x, y in zip(x_blocks, y_blocks):
...         result += x.dot(y)
...     return result
```

```
>>> z = atop(sequence_dot, '', x, 'i', y, 'i')
```

Many dask.array operations are special cases of atop. These tensor operations cover a broad subset of NumPy and this function has been battle tested, supporting tricky concepts like broadcasting.

`dask.array.core.`**`top`**(*func*, *output*, *out_indices*, **arrind_pairs*, ***kwargs*)

Tensor operation

Applies a function, `func`, across blocks from many different input dasks. We arrange the pattern with which those blocks interact with sets of matching indices. E.g.

```
top(func, 'z', 'i', 'x', 'i', 'y', 'i')
```

yield an embarrassingly parallel communication pattern and is read as

$$ z_i = func(x_i, y_i) $$

More complex patterns may emerge, including multiple indices

```
top(func, 'z', 'ij', 'x', 'ij', 'y', 'ji')
```

$$ z_{ij} = func(x_{ij}, y_{ji}) $$

Indices missing in the output but present in the inputs results in many inputs being sent to one function (see examples).

Examples

Simple embarrassing map operation

```
>>> inc = lambda x: x + 1
>>> top(inc, 'z', 'ij', 'x', 'ij', numblocks={'x': (2, 2)})
{('z', 0, 0): (inc, ('x', 0, 0)),
 ('z', 0, 1): (inc, ('x', 0, 1)),
 ('z', 1, 0): (inc, ('x', 1, 0)),
 ('z', 1, 1): (inc, ('x', 1, 1))}
```

Simple operation on two datasets

```
>>> add = lambda x, y: x + y
>>> top(add, 'z', 'ij', 'x', 'ij', 'y', 'ij', numblocks={'x': (2, 2),
...                                                      'y': (2, 2)})
{('z', 0, 0): (add, ('x', 0, 0), ('y', 0, 0)),
 ('z', 0, 1): (add, ('x', 0, 1), ('y', 0, 1)),
 ('z', 1, 0): (add, ('x', 1, 0), ('y', 1, 0)),
 ('z', 1, 1): (add, ('x', 1, 1), ('y', 1, 1))}
```

Operation that flips one of the datasets

```
>>> addT = lambda x, y: x + y.T  # Transpose each chunk
>>> #                                        z_ij ~ x_ij y_ji
>>> #                ..         ..         .. notice swap
>>> top(addT, 'z', 'ij', 'x', 'ij', 'y', 'ji', numblocks={'x': (2, 2),
...                                                       'y': (2, 2)})
{('z', 0, 0): (add, ('x', 0, 0), ('y', 0, 0)),
 ('z', 0, 1): (add, ('x', 0, 1), ('y', 1, 0)),
 ('z', 1, 0): (add, ('x', 1, 0), ('y', 0, 1)),
 ('z', 1, 1): (add, ('x', 1, 1), ('y', 1, 1))}
```

Dot product with contraction over `j` index. Yields list arguments

```
>>> top(dotmany, 'z', 'ik', 'x', 'ij', 'y', 'jk', numblocks={'x': (2, 2),
...                                                         'y': (2, 2)})
{('z', 0, 0): (dotmany, [('x', 0, 0), ('x', 0, 1)],
                        [('y', 0, 0), ('y', 1, 0)]),
 ('z', 0, 1): (dotmany, [('x', 0, 0), ('x', 0, 1)],
                        [('y', 0, 1), ('y', 1, 1)]),
 ('z', 1, 0): (dotmany, [('x', 1, 0), ('x', 1, 1)],
                        [('y', 0, 0), ('y', 1, 0)]),
 ('z', 1, 1): (dotmany, [('x', 1, 0), ('x', 1, 1)],
                        [('y', 0, 1), ('y', 1, 1)])}
```

Supports Broadcasting rules

```
>>> top(add, 'z', 'ij', 'x', 'ij', 'y', 'ij', numblocks={'x': (1, 2),
...                                                      'y': (2, 2)})
{('z', 0, 0): (add, ('x', 0, 0), ('y', 0, 0)),
 ('z', 0, 1): (add, ('x', 0, 1), ('y', 0, 1)),
 ('z', 1, 0): (add, ('x', 0, 0), ('y', 1, 0)),
 ('z', 1, 1): (add, ('x', 0, 1), ('y', 1, 1))}
```

Support keyword arguments with apply

```
>>> def f(a, b=0): return a + b
>>> top(f, 'z', 'i', 'x', 'i', numblocks={'x': (2,), b=10})
{('z', 0): (apply, f, [('x', 0)], {'b': 10}),
 ('z', 1): (apply, f, [('x', 1)], {'b': 10})}
```

Array Methods

class dask.array.**Array**(*dask*, *name*, *chunks*, *dtype=None*, *shape=None*)

Parallel Array

Parameters **dask** : dict

Task dependency graph

name : string

Name of array in dask

shape : tuple of ints

Shape of the entire array

chunks: iterable of tuples

block sizes along each dimension

all(*axis=None*, *keepdims=False*, *split_every=None*)

Test whether all array elements along a given axis evaluate to True.

Parameters **a** : array_like

Input array or object that can be converted to an array.

axis : None or int or tuple of ints, optional

Axis or axes along which a logical AND reduction is performed. The default (*axis = None*) is perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

out : ndarray, 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). See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **all** : ndarray, bool

A new boolean or array is returned unless *out* is specified, in which case a reference to *out* is returned.

See also:

ndarray.all equivalent method

any Test whether any element along a given axis evaluates to True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

Examples

```
>>> np.all([[True,False],[True,True]])
False
```

```
>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False], dtype=bool)
```

```
>>> np.all([-1, 4, 5])
True
```

```
>>> np.all([1.0, np.nan])
True
```

```
>>> o=np.array([False])
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array([ True], dtype=bool))
```

any (*axis=None*, *keepdims=False*, *split_every=None*)

Test whether any array element along a given axis evaluates to True.

Returns single boolean unless *axis* is not `None`

Parameters **a** : array_like

Input array or object that can be converted to an array.

axis : None or int or tuple of ints, optional

Axis or axes along which a logical OR reduction is performed. The default (*axis = None*) is perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

out : ndarray, 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 it is of type float, then it will remain so, returning 1.0 for True and 0.0 for False, regardless of the type of *a*). See *doc.ufuncs* (Section "Output arguments") for details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **any** : bool or ndarray

A new boolean or *ndarray* is returned unless *out* is specified, in which case a reference to *out* is returned.

See also:

ndarray.any equivalent method

all Test whether all elements along a given axis evaluate to True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

Examples

```
>>> np.any([[True, False], [True, True]])
True
```

```
>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False], dtype=bool)
```

```
>>> np.any([-1, 0, 5])
True
```

```
>>> np.any(np.nan)
True
```

```
>>> o=np.array([False])
>>> z=np.any([-1, 4, 5], out=o)
>>> z, o
(array([ True], dtype=bool), array([ True], dtype=bool))
>>> # Check now that z is a reference to o
>>> z is o
True
>>> id(z), id(o) # identity of z and o
(191614240, 191614240)
```

argmax(*axis=None*, *split_every=None*)

Indices of the maximum values along an axis.

Parameters **a** : array_like

Input array.

axis : int, optional

By default, the index is into the flattened array, otherwise along the specified axis.

Returns **index_array** : ndarray of ints

Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed.

See also:

`ndarray.argmax`, *argmin*

amax The maximum value along a given axis.

unravel_index Convert a flat index into an index tuple.

Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

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

```
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])
```

```
>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b) # Only the first occurrence is returned.
1
```

argmin (*axis=None*, *split_every=None*)
Return the indices of the minimum values along an axis.

See also:

argmax Similar function. Please refer to *numpy.argmax* for detailed documentation.

astype (*dtype*, ***kwargs*)
Copy of the array, cast to a specified type

cache (*store=None*, ***kwargs*)
Evaluate and cache array

Parameters store: MutableMapping or ndarray-like

Place to put computed and cached chunks

kwargs:

Keyword arguments to pass on to `get` function for scheduling

Examples

This triggers evaluation and store the result in either

1. An ndarray object supporting setitem (see da.store)
2. A MutableMapping like a dict or chest

It then returns a new dask array that points to this store. This returns a semantically equivalent dask array.

```
>>> import dask.array as da
>>> x = da.arange(5, chunks=2)
>>> y = 2*x + 1
>>> z = y.cache()  # triggers computation
```

```
>>> y.compute()  # Does entire computation
array([1, 3, 5, 7, 9])
```

```
>>> z.compute()  # Just pulls from store
array([1, 3, 5, 7, 9])
```

You might base a cache off of an array like a numpy array or h5py.Dataset.

```
>>> cache = np.empty(5, dtype=x.dtype)
>>> z = y.cache(store=cache)
>>> cache
array([1, 3, 5, 7, 9])
```

Or one might use a MutableMapping like a dict or chest

```
>>> cache = dict()
>>> z = y.cache(store=cache)
>>> cache
{('x', 0): array([1, 3]),
 ('x', 1): array([5, 7]),
 ('x', 2): array([9])}
```

copy()
Copy array. This is a no-op for dask.arrays, which are immutable

cumprod(*axis*, *dtype=None*)
See da.cumprod for docstring

cumsum(*axis*, *dtype=None*)
See da.cumsum for docstring

dot(*a*, *b*, *out=None*)
Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of *a* and the second-to-last of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters **a** : array_like

First argument.

b : array_like

Second argument.

out : ndarray, optional

Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns **output** : ndarray

Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned.

Raises **ValueError**

If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

See also:

vdot Complex-conjugating dot product.

tensordot Sum products over arbitrary axes.

einsum Einstein summation convention.

Examples

```
>>> np.dot(3, 4)
12
```

Neither argument is complex-conjugated:

```
>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

For 2-D arrays it's the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

flatten()

Return a flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Parameters **a** : array_like

Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.

order : {'C','F', 'A', 'K'}, optional

The elements of *a* are read using this index order. 'C' means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

Returns **1d_array** : ndarray

Output of the same dtype as *a*, and of shape `(a.size,)`.

See also:

ndarray.flat 1-D iterator over an array.

ndarray.flatten 1-D array copy of the elements of an array in row-major order.

Notes

In C-like (row-major) order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for Fortran-like, or column-major, index ordering.

Examples

It is equivalent to `reshape(-1, order=order)`.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print np.ravel(x)
[1 2 3 4 5 6]
```

```
>>> print x.reshape(-1)
[1 2 3 4 5 6]
```

```
>>> print np.ravel(x, order='F')
[1 4 2 5 3 6]
```

When `order` is 'A', it will preserve the array's 'C' or 'F' ordering:

```
>>> print np.ravel(x.T)
[1 4 2 5 3 6]
>>> print np.ravel(x.T, order='A')
[1 2 3 4 5 6]
```

When `order` is 'K', it will preserve orderings that are neither 'C' nor 'F', but won't reverse axes:

```
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])
```

```
>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[ 0,  2,  4],
        [ 1,  3,  5]],
       [[ 6,  8, 10],
        [ 7,  9, 11]]])
>>> a.ravel(order='C')
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
>>> a.ravel(order='K')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

map_blocks (*func*, **args*, ***kwargs*)

Map a function across all blocks of a dask array

Parameters **func: callable**

Function to apply to every block in the array

args: dask arrays or constants

dtype: np.dtype

Datatype of resulting array

chunks: tuple (optional)

chunk shape of resulting blocks if the function does not preserve shape

drop_axis: number or iterable (optional)

Dimensions lost by the function

new_axis: number or iterable (optional)

New dimensions created by the function

****kwargs:**

Other keyword arguments to pass to function. Values must be constants (not dask.arrays)

You must also specify the chunks and dtype of the resulting array. If you don't then we assume that the resulting array has the same block structure as the input.

Examples

```
>>> import dask.array as da
>>> x = da.arange(6, chunks=3)
```

```
>>> x.map_blocks(lambda x: x * 2).compute()
array([ 0,  2,  4,  6,  8, 10])
```

The `da.map_blocks` function can also accept multiple arrays

```
>>> d = da.arange(5, chunks=2)
>>> e = da.arange(5, chunks=2)
```

```
>>> f = map_blocks(lambda a, b: a + b**2, d, e)
>>> f.compute()
array([ 0,  2,  6, 12, 20])
```

If function changes shape of the blocks then please provide chunks explicitly.

```
>>> y = x.map_blocks(lambda x: x[::2], chunks=((2, 2),))
```

You have a bit of freedom in specifying chunks. If all of the output chunk sizes are the same, you can provide just that chunk size as a single tuple.

```
>>> a = da.arange(18, chunks=(6,))
>>> b = a.map_blocks(lambda x: x[:3], chunks=(3,))
```

If the function changes the dimension of the blocks you must specify the created or destroyed dimensions.

```
>>> b = a.map_blocks(lambda x: x[None, :, None], chunks=(1, 6, 1),
...                  new_axis=[0, 2])
```

Map_blocks aligns blocks by block positions without regard to shape. In the following example we have two arrays with the same number of blocks but with different shape and chunk sizes.

```
>>> x = da.arange(1000, chunks=(100,))
>>> y = da.arange(100, chunks=(10,))
```

The relevant attribute to match is numblocks

```
>>> x.numblocks
(10,)
>>> y.numblocks
(10,)
```

If these must match (up to broadcasting rules) then we can map arbitrary functions across blocks

```
>>> def func(a, b):
...     return np.array([a.max(), b.max()])
```

```
>>> da.map_blocks(func, x, y, chunks=(2,), dtype='i8')
dask.array<..., shape=(20,), dtype=int64, chunksize=(2,)>
```

```
>>> _.compute()
array([ 99,   9, 199,  19, 299,  29, 399,  39, 499,  49, 599,  59, 699,
        69, 799,  79, 899,  89, 999,  99])
```

Your block function can learn where in the array it is if it supports a `block_id` keyword argument. This will receive entries like (2, 0, 1), the position of the block in the dask array.

```
>>> def func(block, block_id=None):
...     pass
```

You may specify the name of the resulting task in the graph with the optional `name` keyword argument.

```
>>> y = x.map_blocks(lambda x: x + 1, name='increment')
```

map_overlap (*func*, *depth*, *boundary=None*, *trim=True*, ***kwargs*)

Map a function over blocks of the array with some overlap

We share neighboring zones between blocks of the array, then map a function, then trim away the neighboring strips.

Parameters **func: function**

The function to apply to each extended block

depth: int, tuple, or dict

The number of cells that each block should share with its neighbors If a tuple or dict this can be different per axis

boundary: str, tuple, dict

how to handle the boundaries. Values include ‘reflect’, ‘periodic’, ‘nearest’, ‘none’, or any constant value like 0 or np.nan

trim: bool

Whether or not to trim the excess after the map function. Set this to false if your mapping function does this for you.

****kwargs:**

Other keyword arguments valid in `map_blocks`

Examples

```
>>> x = np.array([1, 1, 2, 3, 3, 3, 2, 1, 1])
>>> x = from_array(x, chunks=5)
```

```
>>> def derivative(x):
...     return x - np.roll(x, 1)
```

```
>>> y = x.map_overlap(derivative, depth=1, boundary=0)
>>> y.compute()
array([ 1,  0,  1,  1,  0,  0, -1, -1,  0])
```

```
>>> import dask.array as da
>>> x = np.arange(16).reshape((4, 4))
>>> d = da.from_array(x, chunks=(2, 2))
>>> d.map_overlap(lambda x: x + x.size, depth=1).compute()
array([[16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
>>> func = lambda x: x + x.size
>>> depth = {0: 1, 1: 1}
>>> boundary = {0: 'reflect', 1: 'none'}
>>> d.map_overlap(func, depth, boundary).compute()
array([[12,  13,  14,  15],
       [16,  17,  18,  19],
       [20,  21,  22,  23],
       [24,  25,  26,  27]])
```

max(*axis=None*, *keepdims=False*, *split_every=None*)

Return the maximum of an array or maximum along an axis.

Parameters **a** : array_like

Input data.

axis : int, optional

Axis along which to operate. By default, flattened input is used.

out : ndarray, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **amax** : ndarray or scalar

Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

See also:

amin The minimum value of an array along a given axis, propagating any NaNs.

nanmax The maximum value of an array along a given axis, ignoring any NaNs.

maximum Element-wise maximum of two arrays, propagating any NaNs.

fmax Element-wise maximum of two arrays, ignoring any NaNs.

argmax Return the indices of the maximum values.

nanmin, *minimum*, *fmin*

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding max value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmax.

Don't use *amax* for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `maximum(a[0], a[1])` is faster than `amax(a, axis=0)`.

Examples

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amax(a)           # Maximum of the flattened array
3
>>> np.amax(a, axis=0)   # Maxima along the first axis
array([2, 3])
>>> np.amax(a, axis=1)   # Maxima along the second axis
array([1, 3])
```

```
>>> b = np.arange(5, dtype=np.float)
>>> b[2] = np.NaN
>>> np.amax(b)
nan
>>> np.nanmax(b)
4.0
```

mean(*axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **a** : array_like

Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : data-type, optional

Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : ndarray, optional

Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **m** : ndarray, see dtype parameter above

If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See also:

average Weighted average

std, *var*, *nanmean*, *nanstd*, *nanvar*

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
```

In single precision, *mean* can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.546875
```

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

min(*axis=None*, *keepdims=False*, *split_every=None*)
Return the minimum of an array or minimum along an axis.

Parameters **a** : array_like

Input data.

axis : int, optional

Axis along which to operate. By default, flattened input is used.

out : ndarray, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns amin : ndarray or scalar

Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

See also:

amax The maximum value of an array along a given axis, propagating any NaNs.

nanmin The minimum value of an array along a given axis, ignoring any NaNs.

minimum Element-wise minimum of two arrays, propagating any NaNs.

fmin Element-wise minimum of two arrays, ignoring any NaNs.

argmin Return the indices of the minimum values.

nanmax, *maximum*, *fmax*

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding min value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmin.

Don't use *amin* for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `minimum(a[0], a[1])` is faster than `amin(a, axis=0)`.

Examples

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amin(a)           # Minimum of the flattened array
0
>>> np.amin(a, axis=0)   # Minima along the first axis
array([0, 1])
>>> np.amin(a, axis=1)   # Minima along the second axis
array([0, 2])
```

```
>>> b = np.arange(5, dtype=np.float)
>>> b[2] = np.NaN
>>> np.amin(b)
nan
>>> np.nanmin(b)
0.0
```

moment(*order*, *axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)
Calculate the nth centralized moment.

Parameters **order** : int

Order of the moment that is returned, must be >= 2.

axis : int, optional

Axis along which the central moment is computed. The default is to compute the moment of the flattened array.

dtype : data-type, optional

Type to use in computing the moment. For arrays of integer type the default is float64; for arrays of float types it is the same as the array type.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array.

ddof : int, optional

"Delta Degrees of Freedom": the divisor used in the calculation is N - ddof, where N represents the number of elements. By default ddof is zero.

Returns **moment** : ndarray

References

Computation of Covariances and Arbitrary-Order Statistical Moments" (PDF), Technical Report SAND2008-6212, Sandia National Laboratories

[R151]

nbytes
Number of bytes in array

prod(*axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)
Return the product of array elements over a given axis.

Parameters **a** : array_like

Input data.

axis : None or int or tuple of ints, optional

Axis or axes along which a product is performed. The default (*axis* = *None*) is perform a product over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a product is performed on multiple axes, instead of a single axis or all the axes as before.

dtype : data-type, optional

The data-type of the returned array, as well as of the accumulator in which the elements are multiplied. By default, if *a* is of integer type, *dtype* is the default platform integer. (Note: if the type of *a* is unsigned, then so is *dtype*.) Otherwise, the dtype is the same as that of *a*.

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.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **product_along_axis** : ndarray, see *dtype* parameter above.

An array shaped as *a* but with the specified axis removed. Returns a reference to *out* if specified.

See also:

`ndarray.prod` equivalent method

`numpy.doc.ufuncs` Section "Output arguments"

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

```
>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x) #random
16
```

Examples

By default, calculate the product of all elements:

```
>>> np.prod([1.,2.])
2.0
```

Even when the input array is two-dimensional:

```
>>> np.prod([[1.,2.],[3.,4.]])
24.0
```

But we can also specify the axis over which to multiply:

```
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([  2.,  12.])
```

If the type of *x* is unsigned, then the output type is the unsigned platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True
```

If *x* is of a signed integer type, then the output type is the default platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == np.int
True
```

`ravel`()
Return a flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Parameters **a** : array_like

Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.

order : {'C','F', 'A', 'K'}, optional

The elements of *a* are read using this index order. 'C' means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

Returns **1d_array** : ndarray

Output of the same dtype as *a*, and of shape `(a.size,)`.

See also:

`ndarray.flat` 1-D iterator over an array.

`ndarray.flatten` 1-D array copy of the elements of an array in row-major order.

Notes

In C-like (row-major) order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for Fortran-like, or column-major, index ordering.

Examples

It is equivalent to `reshape(-1, order=order)`.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print np.ravel(x)
[1 2 3 4 5 6]
```

```
>>> print x.reshape(-1)
[1 2 3 4 5 6]
```

```
>>> print np.ravel(x, order='F')
[1 4 2 5 3 6]
```

When `order` is 'A', it will preserve the array's 'C' or 'F' ordering:

```
>>> print np.ravel(x.T)
[1 4 2 5 3 6]
>>> print np.ravel(x.T, order='A')
[1 2 3 4 5 6]
```

When `order` is 'K', it will preserve orderings that are neither 'C' nor 'F', but won't reverse axes:

```
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])
```

```
>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[ 0,  2,  4],
        [ 1,  3,  5]],
       [[ 6,  8, 10],
        [ 7,  9, 11]]])
>>> a.ravel(order='C')
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
>>> a.ravel(order='K')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

rechunk (*chunks*)
See da.rechunk for docstring

reshape (**shape*)
Gives a new shape to an array without changing its data.

Parameters **a** : array_like

Array to be reshaped.

newshape : int or tuple of ints

The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

order : {'C', 'F', 'A'}, optional

Read the elements of *a* using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise.

Returns **reshaped_array** : ndarray

This will be a new view object if possible; otherwise, it will be a copy. Note there is no guarantee of the *memory layout* (C- or Fortran- contiguous) of the returned array.

See also:

ndarray.reshape Equivalent method.

Notes

It is not always possible to change the shape of an array without copying the data. If you want an error to be raise if the data is copied, you should assign the new shape to the shape attribute of the array:

```
>>> a = np.zeros((10, 2))
# A transpose make the array non-contiguous
>>> b = a.T
# Taking a view makes it possible to modify the shape without modifying the
# initial object.
>>> c = b.view()
>>> c.shape = (20)
AttributeError: incompatible shape for a non-contiguous array
```

The *order* keyword gives the index ordering both for *fetching* the values from *a*, and then *placing* the values into the output array. For example, let's say you have an array:

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

You can think of reshaping as first raveling the array (using the given index order), then inserting the elements from the raveled array into the new array using the same kind of index ordering as was used for the raveling.

```
>>> np.reshape(a, (2, 3)) # C-like index ordering
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(np.ravel(a), (2, 3)) # equivalent to C ravel then C reshape
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(a, (2, 3), order='F') # Fortran-like index ordering
array([[0, 4, 3],
       [2, 1, 5]])
>>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
array([[0, 4, 3],
       [2, 1, 5]])
```

Examples

```
>>> a = np.array([[1,2,3], [4,5,6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])
```

```
>>> np.reshape(a, (3,-1))       # the unspecified value is inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

size
Number of elements in array

squeeze()
Remove single-dimensional entries from the shape of an array.

Parameters **a** : array_like

Input data.

axis : None or int or tuple of ints, optional

New in version 1.7.0.

Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

Returns **squeezed** : ndarray

The input array, but with with all or a subset of the dimensions of length 1 removed. This is always *a* itself or a view into *a*.

Examples

```
>>> x = np.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=(2,)).shape
(1, 3)
```

std(*axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)
Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters **a** : array_like

Calculate the standard deviation of these values.

axis : int, optional

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

dtype : dtype, optional

Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

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 calculated values) will be cast if necessary.

ddof : int, optional

Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **standard_deviation** : ndarray, see dtype parameter above.

If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See also:

var, *mean*, *nanmean*, *nanstd*, *nanvar*

numpy.doc.ufuncs Section "Output arguments"

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `std = sqrt(mean(abs(x - x.mean())**2))`.

The average squared deviation is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, *std* takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the *dtype* keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1.,  1.])
>>> np.std(a, axis=1)
array([ 0.5,  0.5])
```

In single precision, std() can be inaccurate:

```
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

store(*target*, ***kwargs*)

Store dask arrays in array-like objects, overwrite data in target

This stores dask arrays into object that supports numpy-style setitem indexing. It stores values chunk by chunk so that it does not have to fill up memory. For best performance you can align the block size of the storage target with the block size of your array.

If your data fits in memory then you may prefer calling `np.array(myarray)` instead.

Parameters **sources: Array or iterable of Arrays**

targets: array-like or iterable of array-likes

These should support setitem syntax `target[10:20] = ...`

lock: boolean or threading.Lock, optional

Whether or not to lock the data stores while storing. Pass True (lock each file individually), False (don't lock) or a particular `threading.Lock` object to be shared among all writes.

compute: boolean, optional

If true compute immediately, return lazy Value object otherwise

Examples

```
>>> x = ...
```

```
>>> import h5py
>>> f = h5py.File('myfile.hdf5')
>>> dset = f.create_dataset('/data', shape=x.shape,
...                                  chunks=x.chunks,
...                                  dtype='f8')
```

```
>>> store(x, dset)
```

Alternatively store many arrays at the same time

```
>>> store([x, y, z], [dset1, dset2, dset3])
```

sum(*axis=None*, *dtype=None*, *keepdims=False*, *split_every=None*)
Sum of array elements over a given axis.

Parameters **a** : array_like

Elements to sum.

axis : None or int or tuple of ints, optional

Axis or axes along which a sum is performed. The default (*axis* = *None*) is perform a sum over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a sum is performed on multiple axes, instead of a single axis or all the axes as before.

dtype : dtype, optional

The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the default platform integer. In that case, the default platform integer is used instead.

out : ndarray, optional

Array into which the output is placed. By default, a new array is created. If *out* is given, it must be of the appropriate shape (the shape of *a* with *axis* removed, i.e., `numpy.delete(a.shape, axis)`). Its type is preserved. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **sum_along_axis** : ndarray

An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

See also:

ndarray.sum Equivalent method.

cumsum Cumulative sum of array elements.

trapz Integration of array values using the composite trapezoidal rule.

mean, average

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
```

If the accumulator is too small, overflow occurs:

```
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

to_delayed()
Convert Array into dask Values

Returns an array of values, one value per chunk.

to_hdf5(*filename*, *datapath*, ***kwargs*)
Store array in HDF5 file

```
>>> x.to_hdf5('myfile.hdf5', '/x')
```

Optionally provide arguments as though to h5py.File.create_dataset

```
>>> x.to_hdf5('myfile.hdf5', '/x', compression='lzf', shuffle=True)
```

See also:

da.store, h5py.File.create_dataset

topk(*k*)
The top k elements of an array

Returns the k greatest elements of the array in sorted order. Only works on arrays of a single dimension.

```
>>> x = np.array([5, 1, 3, 6])
>>> d = from_array(x, chunks=2)
>>> d.topk(2).compute()
array([6, 5])
```

Runs in near linear time, returns all results in a single chunk so all k elements must fit in memory.

transpose(*axes=None*)
Permute the dimensions of an array.

Parameters **a** : array_like

Input array.

axes : list of ints, optional

By default, reverse the dimensions, otherwise permute the axes according to the values given.

Returns **p** : ndarray

a with its axes permuted. A view is returned whenever possible.

See also:

`rollaxis`

Examples

```
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])
```

```
>>> np.transpose(x)
array([[0, 2],
       [1, 3]])
```

```
>>> x = np.ones((1, 2, 3))
>>> np.transpose(x, (1, 0, 2)).shape
(2, 1, 3)
```

var(*axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split_every=None*)
Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters **a** : array_like

Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.

axis : int, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : data-type, optional

Type to use in computing the variance. For arrays of integer type the default is *float32*; for arrays of float types it is the same as the array type.

out : ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddof : int, optional

"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns **variance** : ndarray, see dtype parameter above

If `out=None`, returns a new array containing the variance; otherwise, a reference to the output array is returned.

See also:

std, *mean*, *nanmean*, *nanstd*, *nanvar*

numpy.doc.ufuncs Section "Output arguments"

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

Examples

```
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([ 1.,  1.])
>>> np.var(a, axis=1)
array([ 0.25,  0.25])
```

In single precision, var() can be inaccurate:

```
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096
```

Computing the variance in float64 is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

view(*dtype*, *order='C'*)
Get a view of the array as a new data type

Parameters dtype:

The dtype by which to view the array

order: string

'C' or 'F' (Fortran) ordering

This reinterprets the bytes of the array under a new dtype. If that dtype does not have the same size as the original array then the shape will change.

Beware that both numpy and dask.array can behave oddly when taking shape-changing views of arrays under Fortran ordering. Under some versions of NumPy this function will fail when taking shape-changing views of Fortran ordered arrays if the first dimension has chunks of size one.

vnorm(*ord=None*, *axis=None*, *keepdims=False*, *split_every=None*)
Vector norm

4.4.5 Examples

Creating Dask arrays from NumPy arrays

We can create Dask arrays from any object that implements NumPy slicing, like a `numpy.ndarray` or on-disk formats like h5py or netCDF Dataset objects. This is particularly useful with on disk arrays that don't fit in memory but, for simplicity's sake, we show how this works on a NumPy array.

The following example uses `da.from_array` to create a Dask array from a NumPy array, which isn't particularly valuable (the NumPy array already works in memory just fine) but is easy to play with.

```
>>> import numpy as np
>>> import dask.array as da
>>> x = np.arange(1000)
>>> y = da.from_array(x, chunks=(100))
>>> y.mean().compute()
499.5
```

Creating Dask arrays from HDF5 Datasets

We can construct dask array objects from other array objects that support numpy-style slicing. In this example, we wrap a dask array around an HDF5 dataset, chunking that dataset into blocks of size `(1000, 1000)`:

```
>>> import h5py
>>> f = h5py.File('myfile.hdf5')
>>> dset = f['/data/path']

>>> import dask.array as da
>>> x = da.from_array(dset, chunks=(1000, 1000))
```

Often we have many such datasets. We can use the `stack` or `concatenate` functions to bind many dask arrays into one:

```
>>> dsets = [h5py.File(fn)['/data'] for fn in sorted(glob('myfiles.*.hdf5')]
>>> arrays = [da.from_array(dset, chunks=(1000, 1000)) for dset in dsets]

>>> x = da.stack(arrays, axis=0)   # Stack along a new first axis
```

Note that none of the data is loaded into memory yet, the dask array just contains a graph of tasks showing how to load the data. This allows `dask.array` to do work on datasets that don't fit into RAM.

Creating random arrays

In a simple case, we can create arrays with random data using the `da.random` module.

```
>>> import dask.array as da
>>> x = da.random.normal(0, 1, size=(100000,100000), chunks=(1000, 1000))
>>> x.mean().compute()
-0.0002280808453825202
```

Other topics

4.4.6 Slicing

Dask.array supports most of the NumPy slicing syntax. In particular it supports the following:

- Slicing by integers and slices `x[0, :5]`
- Slicing by lists/arrays of integers `x[[1, 2, 4]]`
- Slicing by lists/arrays of booleans `x[[False, True, True, False, True]]`

It does not currently support the following:

- Slicing one `dask.array` with another `x[x > 0]`
- Slicing with lists in multiple axes `x[[1, 2, 3], [3, 2, 1]]`

Both of these are straightforward to add though. If you have a use case then raise an issue.

Efficiency

The normal dask schedulers are smart enough to compute only those blocks that are necessary to achieve the desired slicing. So large operations may be cheap if only a small output is desired.

In the example below we create a trillion element dask array in million element blocks. We then operate on the entire array and finally slice out only a portion of the output.

```
>>> Trillion element array of ones, in 1000 by 1000 blocks
>>> x = da.ones((1000000, 1000000), chunks=(1000, 1000))

>>> da.exp(x)[:1500, :1500]
...
```

This only needs to compute the top-left four blocks to achieve the result. We are still slightly wasteful on those blocks where we need only partial results. We are also a bit wasteful in that we still need to manipulate the dask-graph with a million or so tasks in it. This can cause an interactive overhead of a second or two.

But generally, slicing works well.

4.4.7 Stack and Concatenate

Often we have many arrays stored on disk that we want to stack together and think of as one large array. This is common with geospatial data in which we might have many HDF5/NetCDF files on disk, one for every day, but we want to do operations that span multiple days.

To solve this problem we use the functions `da.stack` and `da.concatenate`.

Stack

We stack many existing dask Arrays into a new array, creating a new dimension as we go.

```
>>> import dask.array as da
>>> data = [from_array(np.ones((4, 4)), chunks=(2, 2))
...          for i in range(3)]  # A small stack of dask arrays

>>> x = da.stack(data, axis=0)
>>> x.shape
(3, 4, 4)

>>> da.stack(data, axis=1).shape
(4, 3, 4)

>>> da.stack(data, axis=-1).shape
(4, 4, 3)
```

This creates a new dimension with length equal to the number of slices

Concatenate

We concatenate existing arrays into a new array, extending them along an existing dimension

```
>>> import dask.array as da
>>> import numpy as np

>>> data = [from_array(np.ones((4, 4)), chunks=(2, 2))
...          for i in range(3)]  # small stack of dask arrays

>>> x = da.concatenate(data, axis=0)
>>> x.shape
(12, 4)

>>> da.concatenate(data, axis=1).shape
(4, 12)
```

4.4.8 Overlapping Blocks with Ghost Cells

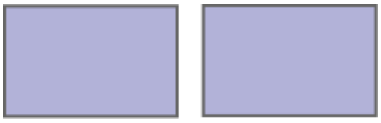
Some array operations require communication of borders between neighboring blocks. Example operations include the following:

- Convolve a filter across an image
- Sliding sum/mean/max, ...
- Search for image motifs like a Gaussian blob that might span the border of a block
- Evaluate a partial derivative
- Play the game of Life

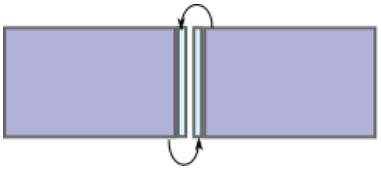
Dask.array supports these operations by creating a new dask.array where each block is slightly expanded by the borders of its neighbors. This costs an excess copy and the communication of many small chunks but allows localized functions to evaluate in an embarrassing manner. We call this process *ghosting*.

Ghosting

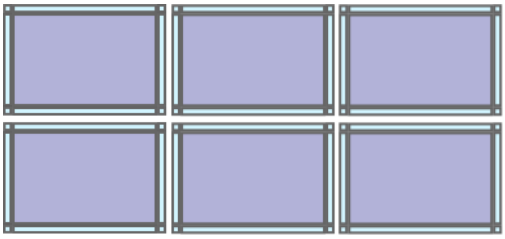
Consider two neighboring blocks in a dask array.



We extend each block by trading thin nearby slices between arrays



We do this in all directions, including also diagonal interactions with the ghost function:



```
>>> import dask.array as da
>>> import numpy as np

>>> x = np.arange(64).reshape((8, 8))
>>> d = da.from_array(x, chunks=(4, 4))
>>> d.chunks
((4, 4), (4, 4))

>>> g = da.ghost.ghost(d, depth={0: 2, 1: 1},
...                       boundary={0: 100, 1: 'reflect'})
>>> g.chunks
((8, 8), (6, 6))
```

```
>>> np.array(g)
array([[100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [  0,   0,   1,   2,   3,   4,   3,   4,   5,   6,   7,   7],
       [  8,   8,   9,  10,  11,  12,  11,  12,  13,  14,  15,  15],
       [ 16,  16,  17,  18,  19,  20,  19,  20,  21,  22,  23,  23],
       [ 24,  24,  25,  26,  27,  28,  27,  28,  29,  30,  31,  31],
       [ 32,  32,  33,  34,  35,  36,  35,  36,  37,  38,  39,  39],
       [ 40,  40,  41,  42,  43,  44,  43,  44,  45,  46,  47,  47],
       [ 16,  16,  17,  18,  19,  20,  19,  20,  21,  22,  23,  23],
       [ 24,  24,  25,  26,  27,  28,  27,  28,  29,  30,  31,  31],
       [ 32,  32,  33,  34,  35,  36,  35,  36,  37,  38,  39,  39],
       [ 40,  40,  41,  42,  43,  44,  43,  44,  45,  46,  47,  47],
       [ 48,  48,  49,  50,  51,  52,  51,  52,  53,  54,  55,  55],
       [ 56,  56,  57,  58,  59,  60,  59,  60,  61,  62,  63,  63],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]])
```

Boundaries

While ghosting you can specify how to handle the boundaries. Current policies include the following:

- `periodic` - wrap borders around to the other side
- `reflect` - reflect each border outwards
- `any-constant` - pad the border with this value

So an example boundary kind argument might look like the following

```
{0: 'periodic',
 1: 'reflect',
 2: np.nan}
```

Alternatively you can use functions like `da.fromfunction` and `da.concatenate` to pad arbitrarily.

Map a function across blocks

Ghosting goes hand-in-hand with mapping a function across blocks. This function can now use the additional information copied over from the neighbors that is not stored locally in each block

```
>>> from scipy.ndimage.filters import gaussian_filter
>>> def func(block):
...     return gaussian_filter(block, sigma=1)

>>> filt = g.map_blocks(func)
```

While in this case we used a SciPy function above this could have been any arbitrary function. This is a good interaction point with Numba.

If your function does not preserve the shape of the block then you will need to provide a `chunks` keyword argument. If your block sizes are regular then this can be a blockshape, such as `(1000, 1000)` or if your blocks are irregular then this must be a full chunks tuple, for example `((1000, 700, 1000), (200, 300))`.

```
>>> g.map_blocks(myfunc, chunks=(5, 5))
```

If your function needs to know the location of the block on which it operates you can give your function a keyword argument `block_id`

```
def func(block, block_id=None):
    ...
```

This extra keyword argument will be given a tuple that provides the block location like `(0, 0)` for the upper right block or `(0, 1)` for the block just to the right of that block.

Trim Excess

After mapping a blocked function you may want to trim off the borders from each block by the same amount by which they were expanded. The function `trim_internal` is useful here and takes the same `depth` argument given to `ghost`.

```
>>> x.chunks
((10, 10, 10, 10), (10, 10, 10, 10))

>>> da.ghost.trim_internal(x, {0: 2, 1: 1})
((6, 6, 6, 6), (8, 8, 8, 8))
```

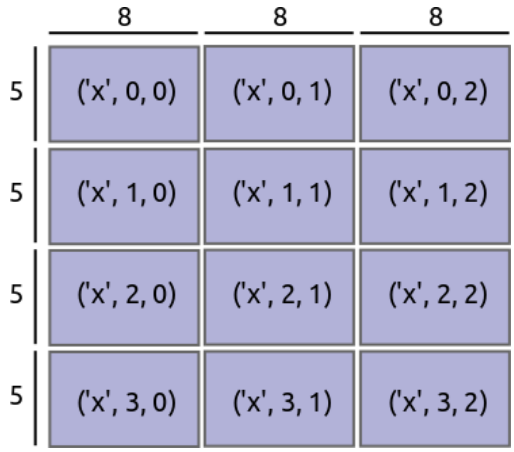
Note: at the moment ‘‘trim‘‘ cuts indiscriminately from the boundaries as well. If you don’t specify a boundary kind then this may not be desired.

Full Workflow

And so a pretty typical ghosting workflow includes `ghost`, `map_blocks`, and `trim_internal`

```
>>> x = ...
>>> g = da.ghost.ghost(x, depth={0: 2, 1: 2},
...                       boundary={0: 'periodic', 1: 'periodic'})
>>> g2 = g.map_blocks(myfunc)
>>> result = da.ghost.trim_internal(g2, {0: 2, 1: 2})
```

4.4.9 Internal Design



Dask arrays define a large array with a grid of blocks of smaller arrays. These arrays may be concrete, or functions that produce arrays. We define a dask array with the following components

- A dask with a special set of keys designating blocks such as `('x', 0, 0), ('x', 0, 1), ...`
- A sequence of chunk sizes along each dimension called `chunks`, for example `((5, 5, 5, 5), (8, 8, 8))`

- A name to identify which keys in the dask refer to this array, like `'x'`

Keys of the dask graph

By special convention we refer to each block of the array with a tuple of the form `(name, i, j, k)` for `i, j, k` being the indices of the block, ranging from `0` to the number of blocks in that dimension. The dask must hold key-value pairs referring to these keys. It likely also holds other key-value pairs required to eventually compute the desired values, for example

```
{
 ('x', 0, 0): (add, 1, ('y', 0, 0)),
 ('x', 0, 1): (add, 1, ('y', 0, 1)),
 ...
 ('y', 0, 0): (getitem, dataset, (slice(0, 1000), slice(0, 1000))),
 ('y', 0, 1): (getitem, dataset, (slice(0, 1000), slice(1000, 2000)))
 ...
}
```

The name of an `Array` object can be found in the `name` attribute. One can get a nested list of keys with the `._keys()` method. One can flatten down this list with `dask.array.core.flatten()`; this is sometimes useful when building new dictionaries.

Chunks

We also store the size of each block along each axis. This is a tuple of tuples such that the length of the outer tuple is equal to the dimension and the lengths of the inner tuples are equal to the number of blocks along each dimension. In the example illustrated above this value is as follows:

```
chunks = ((5, 5, 5, 5), (8, 8, 8))
```

Note that these numbers do not necessarily need to be regular. We often create regularly sized grids but blocks change shape after complex slicing. Beware that some operations do expect certain symmetries in the block-shapes. For example matrix multiplication requires that blocks on each side have anti-symmetric shapes.

Some ways in which `chunks` reflects properties of our array

1. `len(x.chunks) == x.ndim`: The length of chunks is the number of dimensions
2. `tuple(map(sum, x.chunks)) == x.shape`: The sum of each internal chunk, is the length of that dimension.
3. The length of each internal chunk is the number of keys in that dimension. For instance, for `chunks == ((a, b), (d, e, f))` and name `== 'x'` our array has tasks with the following keys:

```
    ('x', 0, 0), ('x', 0, 1), ('x', 0, 2)
    ('x', 1, 0), ('x', 1, 1), ('x', 1, 2)
```

Create an Array Object

So to create an `da.Array` object we need a dictionary with these special keys

```
dsk = {('x', 0, 0): ...}
```

a name specifying to which keys this array refers

```
name = 'x'
```

and a chunks tuple:

```
chunks = ((5, 5, 5, 5), (8, 8, 8))
```

Then one can construct an array:

```
x = da.Array(dsk, name, chunks)
```

So `dask.array` operations update dask dictionaries and track chunks shapes.

Example - `eye` function

As an example lets build the `np.eye` function for `dask.array` to make the identity matrix

```
def eye(n, blocksize):
    chunks = ((blocksize,) * (n // blocksize),
              (blocksize,) * (n // blocksize))

    name = 'eye' + next(tokens)  # unique identifier

    dsk = {(name, i, j): (np.eye, blocksize)
                         if i == j else
                         (np.zeros, (blocksize, blocksize))
              for i in range(n // blocksize)
              for j in range(n // blocksize)}

    dtype = np.eye(0).dtype  # take dtype default from numpy

    return dask.array.Array(dsk, name, chunks, dtype)
```

4.4.10 Extend Dask Array

As discussed in the array design document to create a dask *Array* object you need the following:

1. A dask graph
2. A name specifying a set of keys within that graph
3. A `chunks` tuple giving chunk shape information
4. A `dtype`

Often `dask.array` functions take other `Array` objects as inputs along with parameters, add tasks to a new dask dictionary, create a new `chunks` tuple, and then construct and return a new `Array` object. The hard parts are invariably creating the right tasks and creating a new `chunks` tuple. Careful review of the array design document is suggested.

Example *eye*

Consider this simple example with the `eye` function.

```
from dask.array.core import tokens

def eye(n, blocksize):
    chunks = ((blocksize,) * n // blocksize,
              (blocksize,) * n // blocksize)
```

```
    name = 'eye' + next(tokens)  # unique identifier

    dsk = {(name, i, j): (np.eye, blocksize)
                         if i == j else
                         (np.zeros, (blocksize, blocksize))
           for i in range(n // blocksize)
           for j in range(n // blocksize)}

    dtype = np.eye(0).dtype  # take dtype default from numpy

    return Array(dsk, name, chunks, dtype)
```

This example is particularly simple because it doesn't take any `Array` objects as input.

Example *diag*

Consider the function `diag` that takes a 1d vector and produces a 2d matrix with the values of the vector along the diagonal. Consider the case where the input is a 1d array with chunk sizes `(2, 3, 4)` in the first dimension like this:

```
[x_0, x_1], [x_2, x_3, x_4], [x_5, x_6, x_7, x_8]
```

We need to create a 2d matrix with chunks equal to `((2, 3, 4), (2, 3, 4))` where the ith block along the diagonal of the output is the result of calling `np.diag` on the `ith` block of the input and all other blocks are zero.

```
def diag(v):
    """Construct a diagonal array, with ``v`` on the diagonal."""
    chunks = (v.chunks[0], v.chunks[0])  # repeat chunks twice

    name = 'diag' + next(tokens)  # unique identifier

    dsk = {(name, i, j): (np.diag, (v.name, i))
                         if i == j else
                         (np.zeros, (v.chunks[0][i], v.chunks[0][j]))
           for i in range(len(v.chunks[0]))
           for j in range(len(v.chunks[0]))}

    dsk.update(v.dask)  # include dask graph of the input

    dtype = v.dtype     # output has the same dtype as the input

    return Array(dsk, name, chunks, dtype)

>>> x = da.arange(9, chunks=((2, 3, 4),))
>>> x
dask.array<arange-1, shape=(9,), chunks=((2, 3, 4)), dtype=int64>

>>> M = diag(x)
>>> M
dask.array<diag-2, shape=(9, 9), chunks=((2, 3, 4), (2, 3, 4)), dtype=int64>

>>> M.compute()
array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 2, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 3, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 4, 0, 0, 0, 0],
```

```
        [0, 0, 0, 0, 0, 5, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 6, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 7, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 8]])
```

4.4.11 LinearOperator

Dask arrays implement the SciPy LinearOperator interface and so can be used with any SciPy algorithm depending on that interface.

Example

```
import dask.array as da
x = da.random.random(size=(10000, 10000), chunks=(1000, 1000))

from scipy.sparse.linalg.interface import MatrixLinearOperator
A = MatrixLinearOperator(x)

import numpy as np
b = np.random.random(10000)

from scipy.sparse.linalg import gmres
x = gmres(A, b)
```

Disclaimer: This is just a toy example and not necessarily the best way to solve this problem for this data.

4.5 Bag

Dask.Bag parallelizes computations across a large collection of generic Python objects. It is particularly useful when dealing with large quantities of semi-structured data like JSON blobs or log files.

4.5.1 Overview

Dask Bag implements a operations like `map`, `filter`, `fold`, `frequencies` and `groupby` on lists of Python objects. It does this in parallel using multiple processes and in small memory using Python iterators building off of libraries like PyToolz.

Design

Dask bags coordinate many Python lists or Iterators, each of which forms a partition of a larger linear collection.

Common Uses

Dask bags are often used to parallelize simple computations on unstructured or semi-structured data like text data, log files, JSON records, or user defined Python objects.

Execution

Execution on bags provide two benefits:

1. Streaming: data processes lazily, allowing smooth execution of larger-than-memory data
2. Parallel: data is split up, allowing multiple cores to execute in parallel.

Default scheduler

By default `dask.bag` uses `dask.multiprocessing` for computation. As a benefit dask bypasses the GIL and uses multiple cores on Pure Python objects. As a drawback dask.bag doesn't perform well on computations that include a great deal of inter-worker communication. For common operations this is rarely an issue as most `dask.bag` workflows are embarrassingly parallel or result in reductions with little data moving between workers.

Shuffle

Some operations, like full `groupby` and bag-to-bag `join` do require substantial inter-worker communication. These are handled specially by shuffle operations that use disk and a central memory server as a central point of communication.

Shuffle operations are expensive and better handled by projects like `dask.dataframe`. It is best to use `dask.bag` to clean and process data, then transform it into an array or dataframe before embarking on the more complex operations that require shuffle steps.

Dask uses partd to perform efficient, parallel, spill-to-disk shuffles.

Known Limitations

Bags provide very general computation (any Python function.) This generality comes at cost. Bags have the following known limitations:

1. By default they rely on the multiprocessing scheduler, which has its own set of known limitations (see Shared Memory)
2. Bag operations tend to be slower than array/dataframe computations in the same way that Python tends to be slower than NumPy/pandas
3. `Bag.groupby` is slow. You should try to use `Bag.foldby` if possible. Using `Bag.foldby` requires more thought.
4. The implementation backing `Bag.groupby` is under heavy churn.

Name

Bag is the mathematical name for an unordered collection allowing repeats. It is a friendly synonym to multiset. A bag or a multiset is a generalization of the concept of a set that, unlike a set, allows multiple instances of the multiset's elements.

- `list`: *ordered* collection *with repeats*, `[1, 2, 3, 2]`
- `set`: *unordered* collection *without repeats*, `{1, 2, 3}`
- `bag`: *unordered* collection *with repeats*, `{1, 2, 2, 3}`

So a bag is like a list, but it doesn't guarantee an ordering among elements. There can be repeated elements but you can't ask for the ith element.

4.5.2 Create Dask Bags

There are several ways to create dask.bags around your data:

db.from_sequence

You can create a bag from an existing Python sequence:

```
>>> import dask.bag as db
>>> b = db.from_sequence([1, 2, 3, 4, 5, 6])
```

You can control the number of partitions into which this data is binned:

```
>>> b = db.from_sequence([1, 2, 3, 4, 5, 6], npartitions=2)
```

This controls the granularity of the parallelism that you expose. By default dask will try to partition your data into about 100 partitions.

IMPORTANT: do not load your data into Python and then load that data into dask.bag. Instead, use dask.bag to load your data. This parallelizes the loading step and reduces inter-worker communication:

```
>>> b = db.from_sequence(['1.dat', '2.dat', ...]).map(load_from_filename)
```

db.read_text

Dask.bag can load data directly from textfiles. You can pass either a single filename, a list of filenames, or a globstring. The resulting bag will have one item per line, one file per partition:

```
>>> b = db.read_text('myfile.json')
>>> b = db.read_text(['myfile.1.json', 'myfile.2.json', ...])
>>> b = db.read_text('myfile.*.json')
```

This handles standard compression libraries like `gzip`, `bz2`, `xz`, or any easily installed compression library that has a File-like object. Compression will be inferred by filename extension, or by using the `compression='gzip'` keyword:

```
>>> b = db.read_text('myfile.*.json.gz')
```

The resulting items in the bag are strings. You may want to parse them using functions like `json.loads`:

```
>>> import json
>>> b = db.read_text('myfile.*.json.gz', compression='gzip').map(json.loads)
```

Or do string munging tasks. For convenience there is a string namespace attached directly to bags with `.str.methodname`:

```
>>> b = db.read_text('myfile.*.csv.gz').str.strip().str.split(',')
```

db.from_delayed

You can construct a dask bag from dask.delayed values using the `db.from_delayed` function. See documentation on using dask.delayed with collections for more information.

4.5.3 Store Dask Bags

In Memory

You can convert a dask bag to a list or Python iterable by calling `compute()` or by converting the object into a list

```
>>> result = b.compute()
or
>>> result = list(b)
```

To Textfiles

You can convert a dask bag into a sequence of files on disk by calling the `.to_textfiles()` method

`dask.bag.core.`**`to_textfiles`**(*b*, *path*, *name_function=None*, *compression='infer'*, *encoding='utf-8'*, *compute=True*)

Write bag to disk, one filename per partition, one line per element

Paths: This will create one file for each partition in your bag. You can specify the filenames in a variety of ways.

Use a globstring

```
>>> b.to_textfiles('/path/to/data/*.json.gz')
```

The * will be replaced by the increasing sequence 1, 2, ...

```
/path/to/data/0.json.gz
/path/to/data/1.json.gz
```

Use a globstring and a `name_function=` keyword argument. The name_function function should expect an integer and produce a string. Strings produced by name_function must preserve the order of their respective partition indices.

```
>>> from datetime import date, timedelta
>>> def name(i):
...     return str(date(2015, 1, 1) + i * timedelta(days=1))
```

```
>>> name(0)
'2015-01-01'
>>> name(15)
'2015-01-16'
```

```
>>> b.to_textfiles('/path/to/data/*.json.gz', name_function=name)
```

```
/path/to/data/2015-01-01.json.gz
/path/to/data/2015-01-02.json.gz
...
```

You can also provide an explicit list of paths.

```
>>> paths = ['/path/to/data/alice.json.gz', '/path/to/data/bob.json.gz', ...]
>>> b.to_textfiles(paths)
```

Compression: Filenames with extensions corresponding to known compression algorithms (gz, bz2) will be compressed accordingly.

Bag Contents: The bag calling `to_textfiles` must be a bag of text strings. For example, a bag of dictionaries could be written to JSON text files by mapping `json.dumps` on to the bag first, and then calling `to_textfiles`:

```
>>> b_dict.map(json.dumps).to_textfiles("/path/to/data/*.json")
```

To DataFrames

You can convert a dask bag into a dask dataframe and use those storage solutions.

`Bag.`**`to_dataframe`**(*columns=None*)
Convert Bag to dask.dataframe

Bag should contain tuple or dict records.

Provide `columns=` keyword arg to specify column names.

Index will not be particularly meaningful. Use `reindex` afterwards if necessary.

Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence([{'name': 'Alice',   'balance': 100},
...                       {'name': 'Bob',     'balance': 200},
...                       {'name': 'Charlie', 'balance': 300}],
...                      npartitions=2)
>>> df = b.to_dataframe()
```

```
>>> df.compute()
   balance     name
0      100    Alice
1      200      Bob
0      300  Charlie
```

To Delayed Values

You can convert a dask bag into a list of dask delayed values and custom storage solutions from there.

`Bag.`**`to_delayed`**()
Convert bag to dask Values

Returns list of values, one value per partition.

4.5.4 API

Top level user functions:

Bag(dsk, name, npartitions)	Parallel collection of Python objects
Bag.all((iterable) -> bool)	Return True if bool(x) is True for all values x in the iterable.
Bag.any((iterable) -> bool)	Return True if bool(x) is True for any x in the iterable.
Bag.compute(**kwargs)	Compute several dask collections at once.
Bag.concat()	Concatenate nested lists into one long list
Bag.count([split_every])	Count the number of elements
Bag.distinct()	Distinct elements of collection
Bag.filter(predicate)	Filter elements in collection by a predicate function
Bag.fold(binop[, combine, initial, split_every])	Parallelizable reduction

Table 4.7 – continued from previous page

Bag.foldby(key, binop[, initial, combine, ...])	Combined reduction and groupby
Bag.frequencies([split_every])	Count number of occurrences of each distinct element
Bag.groupby(grouper[, method, npartitions, ...])	Group collection by key function
Bag.join(other, on_self[, on_other])	Join collection with another collection
Bag.map(func, **kwargs)	Map a function across all elements in collection
Bag.map_partitions(func, **kwargs)	Apply function to every partition within collection
Bag.max((iterable[[, key])	max(a, b, c, ...[, key=func]) -> value
Bag.mean()	Arithmetic mean
Bag.min((iterable[[, key])	min(a, b, c, ...[, key=func]) -> value
Bag.pluck(key[, default])	Select item from all tuples/dicts in collection
Bag.product(other)	Cartesian product between two bags
Bag.reduction(perpartition, aggregate[, ...])	Reduce collection with reduction operators
Bag.random_sample(prob[, random_state])	Return elements from bag with probability of `prob`.
Bag.remove(predicate)	Remove elements in collection that match predicate
Bag.repartition(npartitions)	Coalesce bag into fewer partitions
Bag.std([ddof])	Standard deviation
Bag.sum((sequence[, start]) -> value)	Return the sum of a sequence of numbers (NOT strings) plus the value of p
Bag.take(k[, compute])	Take the first k elements
Bag.to_dataframe([columns])	Convert Bag to dask.dataframe
Bag.to_delayed()	Convert bag to dask Values
Bag.to_textfiles(path[, name_function, ...])	Write bag to disk, one filename per partition, one line per element
Bag.topk(k[, key, split_every])	K largest elements in collection
Bag.var([ddof])	Variance
`Bag.visualize`([filename, format, optimize_graph])	Render the computation of this object's task graph using graphviz.

Create Bags

from_sequence(seq[, partition_size, npartitions])	Create dask from Python sequence
from_delayed(values)	Create bag from many dask.delayed objects
read_text(urlpath[, blocksize, compression, ...])	Read lines from text files
from_castra(x[, columns, index])	Load a dask Bag from a Castra.
from_url(urls)	Create a dask.bag from a url
range(n, npartitions)	Numbers from zero to n
concat(bags)	Concatenate many bags together, unioning all elements
zip(*bags)	Partition-wise bag zip

Turn Bags into other things

Bag.to_textfiles(path[, name_function, ...])	Write bag to disk, one filename per partition, one line per element
Bag.to_dataframe([columns])	Convert Bag to dask.dataframe

Bag methods

class `dask.bag.`**`Bag`**(*dsk*, *name*, *npartitions*)

Parallel collection of Python objects

Examples

Create Bag from sequence

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.filter(lambda x: x % 2 == 0).map(lambda x: x * 10))
[0, 20, 40]
```

Create Bag from filename or globstring of filenames

```
>>> b = db.read_text('/path/to/mydata.*.json.gz').map(json.loads)
```

Create manually (expert use)

```
>>> dsk = {('x', 0): (range, 5),
...        ('x', 1): (range, 5),
...        ('x', 2): (range, 5)}
>>> b = Bag(dsk, 'x', npartitions=3)
```

```
>>> sorted(b.map(lambda x: x * 10))
[0, 0, 0, 10, 10, 10, 20, 20, 20, 30, 30, 30, 40, 40, 40]
```

```
>>> int(b.fold(lambda x, y: x + y))
30
```

accumulate(*binop*, *initial='__no__default__'*)
Repeatedly apply binary function to a sequence, accumulating results.

Examples

```
>>> from operator import add
>>> b = from_sequence([1, 2, 3, 4, 5], npartitions=2)
>>> b.accumulate(add).compute()
[1, 3, 6, 10, 15]
```

Accumulate also takes an optional argument that will be used as the first value.

```
>>> b.accumulate(add, -1)
[-1, 0, 2, 5, 9, 15]
```

all(*iterable*) → bool
Return True if bool(x) is True for all values x in the iterable. If the iterable is empty, return True.

any(*iterable*) → bool
Return True if bool(x) is True for any x in the iterable. If the iterable is empty, return False.

concat()
Concatenate nested lists into one long list

```
>>> b = from_sequence([[1], [2, 3]])
>>> list(b)
[[1], [2, 3]]
```

```
>>> list(b.concat())
[1, 2, 3]
```

count(*split_every=None*)
Count the number of elements

distinct()
Distinct elements of collection

Unordered without repeats.

```
>>> b = from_sequence(['Alice', 'Bob', 'Alice'])
>>> sorted(b.distinct())
['Alice', 'Bob']
```

filter(*predicate*)
Filter elements in collection by a predicate function

```
>>> def iseven(x):
...     return x % 2 == 0
```

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.filter(iseven))
[0, 2, 4]
```

fold(*binop*, *combine=None*, *initial='__no__default__'*, *split_every=None*)
Parallelizable reduction

Fold is like the builtin function `reduce` except that it works in parallel. Fold takes two binary operator functions, one to reduce each partition of our dataset and another to combine results between partitions

1. `binop`: Binary operator to reduce within each partition
2. `combine`: Binary operator to combine results from binop

Sequentially this would look like the following:

```
>>> intermediates = [reduce(binop, part) for part in partitions]
>>> final = reduce(combine, intermediates)
```

If only one function is given then it is used for both functions `binop` and `combine` as in the following example to compute the sum:

```
>>> def add(x, y):
...     return x + y
```

```
>>> b = from_sequence(range(5))
>>> b.fold(add).compute()
10
```

In full form we provide both binary operators as well as their default arguments

```
>>> b.fold(binop=add, combine=add, initial=0).compute()
10
```

More complex binary operators are also doable

```
>>> def add_to_set(acc, x):
...     ''' Add new element x to set acc '''
...     return acc | set([x])
>>> b.fold(add_to_set, set.union, initial=set()).compute()
{1, 2, 3, 4, 5}
```

See also:

Bag.foldby

foldby(*key*, *binop*, *initial='__no__default__'*, *combine=None*, *combine_initial='__no__default__'*)
Combined reduction and groupby

Foldby provides a combined groupby and reduce for efficient parallel split-apply-combine tasks.

The computation

```
>>> b.foldby(key, binop, init)
```

is equivalent to the following:

```
>>> def reduction(group):
...     return reduce(binop, group, init)
```

```
>>> b.groupby(key).map(lambda (k, v): (k, reduction(v)))
```

But uses minimal communication and so is *much* faster.

```
>>> b = from_sequence(range(10))
>>> iseven = lambda x: x % 2 == 0
>>> add = lambda x, y: x + y
>>> dict(b.foldby(iseven, add))
{True: 20, False: 25}
```

Key Function

The key function determines how to group the elements in your bag. In the common case where your bag holds dictionaries then the key function often gets out one of those elements.

```
>>> def key(x):
...     return x['name']
```

This case is so common that it is special cased, and if you provide a key that is not a callable function then dask.bag will turn it into one automatically. The following are equivalent:

```
>>> b.foldby(lambda x: x['name'], ...)
>>> b.foldby('name', ...)
```

Binops

It can be tricky to construct the right binary operators to perform analytic queries. The `foldby` method accepts two binary operators, `binop` and `combine`. Binary operators two inputs and output must have the same type.

Binop takes a running total and a new element and produces a new total:

```
>>> def binop(total, x):
...     return total + x['amount']
```

Combine takes two totals and combines them:

```
>>> def combine(total1, total2):
...     return total1 + total2
```

Each of these binary operators may have a default first value for total, before any other value is seen. For addition binary operators like above this is often `0` or the identity element for your operation.

```
>>> b.foldby('name', binop, 0, combine, 0)
```

See also:

`toolz.reduceby`, `pyspark.combineByKey`

frequencies(*split_every=None*)
Count number of occurrences of each distinct element

```
>>> b = from_sequence(['Alice', 'Bob', 'Alice'])
>>> dict(b.frequencies())
{'Alice': 2, 'Bob', 1}
```

groupby(*grouper*, *method=None*, *npartitions=None*, *blocksize=1048576*, *max_branch=None*)
Group collection by key function

This requires a full dataset read, serialization and shuffle. This is expensive. If possible you should use `foldby`.

Parameters grouper: function

Function on which to group elements

method: str

Either 'disk' for an on-disk shuffle or 'tasks' to use the task scheduling framework. Use 'disk' if you are on a single machine and 'tasks' if you are on a distributed cluster.

npartitions: int

If using the disk-based shuffle, the number of output partitions

blocksize: int

If using the disk-based shuffle, the size of shuffle blocks

max_branch: int

If using the task-based shuffle, the amount of splitting each partition undergoes. Increase this for fewer copies but more scheduler overhead.

See also:

Bag.foldby

Examples

```
>>> b = from_sequence(range(10))
>>> iseven = lambda x: x % 2 == 0
>>> dict(b.groupby(iseven))
{True: [0, 2, 4, 6, 8], False: [1, 3, 5, 7, 9]}
```

join(*other*, *on_self*, *on_other=None*)
Join collection with another collection

Other collection must be an Iterable, and not a Bag.

```
>>> people = from_sequence(['Alice', 'Bob', 'Charlie'])
>>> fruit = ['Apple', 'Apricot', 'Banana']
>>> list(people.join(fruit, lambda x: x[0]))
[('Apple', 'Alice'), ('Apricot', 'Alice'), ('Banana', 'Bob')]
```

map(*func*, ***kwargs*)
Map a function across all elements in collection

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.map(lambda x: x * 10))
[0, 10, 20, 30, 40]
```

Keyword arguments are passed through to `func`. These can be either `dask.bag.Item`, or normal python objects.

Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(1, 101), npartitions=10)
>>> def div(num, den=1):
...     return num / den
```

Using a python object:

```
>>> hi = b.max().compute()
>>> hi
100
>>> b.map(div, den=hi).take(5)
(0.01, 0.02, 0.03, 0.04, 0.05)
```

Using an `Item`:

```
>>> b.map(div, den=b.max()).take(5)
(0.01, 0.02, 0.03, 0.04, 0.05)
```

Note that while both versions give the same output, the second forms a single graph, and then computes everything at once, and in some cases may be more efficient.

map_partitions (*func*, ***kwargs*)
Apply function to every partition within collection

Note that this requires you to understand how dask.bag partitions your data and so is somewhat internal.

```
>>> b.map_partitions(myfunc)
```

Keyword arguments are passed through to `func`. These can be either `dask.bag.Item`, or normal python objects.

Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(1, 101), npartitions=10)
>>> def div(nums, den=1):
...     return [num / den for num in nums]
```

Using a python object:

```
>>> hi = b.max().compute()
>>> hi
100
>>> b.map_partitions(div, den=hi).take(5)
(0.01, 0.02, 0.03, 0.04, 0.05)
```

Using an `Item`:

```
>>> b.map_partitions(div, den=b.max()).take(5)
(0.01, 0.02, 0.03, 0.04, 0.05)
```

Note that while both versions give the same output, the second forms a single graph, and then computes everything at once, and in some cases may be more efficient.

max (*iterable*[, *key=func*]) → value
max(a, b, c, ...[, key=func]) -> value

With a single iterable argument, return its largest item. With two or more arguments, return the largest argument.

mean ()
Arithmetic mean

min (*iterable*[, *key=func*]) → value
min(a, b, c, ...[, key=func]) -> value

With a single iterable argument, return its smallest item. With two or more arguments, return the smallest argument.

pluck (*key*, *default='__no__default__'*)
Select item from all tuples/dicts in collection

```
>>> b = from_sequence([{'name': 'Alice', 'credits': [1, 2, 3]},
...                    {'name': 'Bob',   'credits': [10, 20]}])
>>> list(b.pluck('name'))
['Alice', 'Bob']
>>> list(b.pluck('credits').pluck(0))
[1, 10]
```

product (*other*)
Cartesian product between two bags

random_sample (*prob*, *random_state=None*)
Return elements from bag with probability of `prob`.

`prob` must be a number in the interval *[0, 1]*. All elements are considered independently without replacement.

Providing an integer seed for `random_state` will result in deterministic sampling. Given the same seed it will return the same sample every time.

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.random_sample(0.5, 42))
[1, 3]
>>> list(b.random_sample(0.5, 42))
[1, 3]
```

reduction (*perpartition*, *aggregate*, *split_every=None*, *out_type=<class 'dask.bag.core.Item'>*, *name=None*)
Reduce collection with reduction operators

Parameters **perpartition: function**

reduction to apply to each partition

aggregate: function

reduction to apply to the results of all partitions

split_every: int (optional)

Group partitions into groups of this size while performing reduction Defaults to 8

out_type: {Bag, Item}

The out type of the result, Item if a single element, Bag if a list of elements. Defaults to Item.

Examples

```
>>> b = from_sequence(range(10))
>>> b.reduction(sum, sum).compute()
45
```

remove (*predicate*)
Remove elements in collection that match predicate

```
>>> def iseven(x):
...     return x % 2 == 0
```

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.remove(iseven))
[1, 3]
```

repartition (*npartitions*)
Coalesce bag into fewer partitions

Examples

```
>>> b.repartition(5)  # set to have 5 partitions
```

std (*ddof=0*)
Standard deviation

sum (*sequence*[, *start*]) → value
Return the sum of a sequence of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the sequence is empty, return start.

take (*k*, *compute=True*)
Take the first k elements

Evaluates by default, use `compute=False` to avoid computation. Only takes from the first partition

```
>>> b = from_sequence(range(10))
>>> b.take(3)
(0, 1, 2)
```

to_dataframe (*columns=None*)
Convert Bag to dask.dataframe

Bag should contain tuple or dict records.

Provide `columns=` keyword arg to specify column names.

Index will not be particularly meaningful. Use `reindex` afterwards if necessary.

Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence([{'name': 'Alice',   'balance': 100},
...                       {'name': 'Bob',     'balance': 200},
...                       {'name': 'Charlie', 'balance': 300}],
...                      npartitions=2)
>>> df = b.to_dataframe()
```

```
>>> df.compute()
   balance     name
0      100    Alice
1      200      Bob
0      300  Charlie
```

to_delayed()
Convert bag to dask Values

Returns list of values, one value per partition.

to_textfiles(*path*, *name_function=None*, *compression='infer'*, *encoding='utf-8'*, *compute=True*)
Write bag to disk, one filename per partition, one line per element

Paths: This will create one file for each partition in your bag. You can specify the filenames in a variety of ways.

Use a globstring

```
>>> b.to_textfiles('/path/to/data/*.json.gz')
```

The * will be replaced by the increasing sequence 1, 2, ...

```
/path/to/data/0.json.gz
/path/to/data/1.json.gz
```

Use a globstring and a `name_function=` keyword argument. The name_function function should expect an integer and produce a string. Strings produced by name_function must preserve the order of their respective partition indices.

```
>>> from datetime import date, timedelta
>>> def name(i):
...     return str(date(2015, 1, 1) + i * timedelta(days=1))
```

```
>>> name(0)
'2015-01-01'
>>> name(15)
'2015-01-16'
```

```
>>> b.to_textfiles('/path/to/data/*.json.gz', name_function=name)
```

```
/path/to/data/2015-01-01.json.gz
/path/to/data/2015-01-02.json.gz
...
```

You can also provide an explicit list of paths.

```
>>> paths = ['/path/to/data/alice.json.gz', '/path/to/data/bob.json.gz', ...]
>>> b.to_textfiles(paths)
```

Compression: Filenames with extensions corresponding to known compression algorithms (gz, bz2) will be compressed accordingly.

Bag Contents: The bag calling `to_textfiles` must be a bag of text strings. For example, a bag of dictionaries could be written to JSON text files by mapping `json.dumps` on to the bag first, and then calling `to_textfiles` :

```
>>> b_dict.map(json.dumps).to_textfiles("/path/to/data/*.json")
```

topk(*k*, *key=None*, *split_every=None*)
K largest elements in collection

Optionally ordered by some key function

```
>>> b = from_sequence([10, 3, 5, 7, 11, 4])
>>> list(b.topk(2))
[11, 10]
```

```
>>> list(b.topk(2, lambda x: -x))
[3, 4]
```

unzip(*n*)
Transform a bag of tuples to `n` bags of their elements.

Examples

```
>>> b = from_sequence([(i, i + 1, i + 2) for i in range(10)])
>>> first, second, third = b.unzip(3)
>>> isinstance(first, Bag)
True
>>> first.compute()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that this is equivalent to:

```
>>> first, second, third = (b.pluck(i) for i in range(3))
```

var(*ddof=0*)
Variance

Other functions

dask.bag.**from_sequence**(*seq*, *partition_size=None*, *npartitions=None*)
Create dask from Python sequence

This sequence should be relatively small in memory. Dask Bag works best when it handles loading your data itself. Commonly we load a sequence of filenames into a Bag and then use `.map` to open them.

Parameters **seq: Iterable**

A sequence of elements to put into the dask

partition_size: int (optional)

The length of each partition

npartitions: int (optional)

The number of desired partitions

It is best to provide either ‘‘partition_size‘‘ or ‘‘npartitions‘‘

(though not both.)

See also:

read_text Create bag from textfiles

Examples

```
>>> b = from_sequence(['Alice', 'Bob', 'Chuck'], partition_size=2)
```

dask.bag.**from_delayed**(*values*)

Create bag from many dask.delayed objects

Parameters **values: list of Values**

An iterable of dask.delayed.Value objects, such as come from dask.do These comprise the individual partitions of the resulting bag

Returns Bag

Examples

```
>>> b = from_delayed([x, y, z])
```

dask.bag.**read_text**(*urlpath*, *blocksize=None*, *compression='infer'*, *encoding='utf-8'*, *errors='strict'*, *linedelimiter='\n'*, *collection=True*, *storage_options=None*)

Read lines from text files

Parameters **urlpath: string or list**

Absolute or relative filepath, URL (may include protocols like `s3://`), globstring, or a list of beforementioned strings.

blocksize: None or int

Size to cut up larger files. Streams by default.

compression: string

Compression format like 'gzip' or 'xz'. Defaults to 'infer'

encoding: string

errors: string

linedelimiter: string

collection: bool, optional

Return dask.bag if True, or list of delayed values if false

storage_options: dict

Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns dask.bag.Bag if collection is True or list of Delayed lists otherwise

See also:

from_sequence Build bag from Python sequence

Examples

```
>>> b = read_text('myfiles.1.txt')
>>> b = read_text('myfiles.*.txt')
>>> b = read_text('myfiles.*.txt.gz')
>>> b = read_text('s3://bucket/myfiles.*.txt')
>>> b = read_text('s3://key:secret@bucket/myfiles.*.txt')
>>> b = read_text('hdfs://namenode.example.com/myfiles.*.txt')
```

Parallelize a large file by providing the number of uncompressed bytes to load into each partition.

```
>>> b = read_text('largefile.txt', blocksize=1e7)
```

dask.bag.**from_castra**(*x*, *columns=None*, *index=False*)

Load a dask Bag from a Castra.

Parameters **x** : filename or Castra

columns: list or string, optional

The columns to load. Default is all columns.

index: bool, optional

If True, the index is included as the first element in each tuple. Default is False.

dask.bag.**from_url**(*urls*)

Create a dask.bag from a url

```
>>> a = from_url('http://raw.githubusercontent.com/dask/dask/master/README.rst')
>>> a.npartitions
1
```

```
>>> a.take(8)
('Dask\n',
 '====\n',
 '\n',
 '|Build Status| |Coverage| |Doc Status| |Gitter|\n',
 '\n',
 'Dask provides multi-core execution on larger-than-memory datasets using blocked\n',
 'algorithms and task scheduling.  It maps high-level NumPy and list operations\n',
 'on large datasets on to graphs of many operations on small in-memory datasets.\n')
```

```
>>> b = from_url(['http://github.com', 'http://google.com'])
>>> b.npartitions
2
```

dask.bag.**range**(*n*, *npartitions*)

Numbers from zero to n

Examples

```
>>> import dask.bag as db
>>> b = db.range(5, npartitions=2)
>>> list(b)
[0, 1, 2, 3, 4]
```

dask.bag.**concat**(*bags*)

Concatenate many bags together, unioning all elements

```
>>> import dask.bag as db
>>> a = db.from_sequence([1, 2, 3])
>>> b = db.from_sequence([4, 5, 6])
>>> c = db.concat([a, b])
```

```
>>> list(c)
[1, 2, 3, 4, 5, 6]
```

dask.bag.**zip**(**bags*)

Partition-wise bag zip

All passed bags must have the same number of partitions.

NOTE: corresponding partitions should have the same length; if they do not, the "extra" elements from the longer partition(s) will be dropped. If you have this case chances are that what you really need is a data alignment mechanism like pandas's, and not a missing value filler like zip_longest.

Examples

Correct usage:

```
>>> import dask.bag as db
>>> evens = db.from_sequence(range(0, 10, 2), partition_size=4)
>>> odds = db.from_sequence(range(1, 10, 2), partition_size=4)
>>> pairs = db.zip(evens, odds)
>>> list(pairs)
[(0, 1), (2, 3), (4, 5), (6, 7), (8, 9)]
```

Incorrect usage:

```
>>> numbers = db.range(20)
>>> fizz = numbers.filter(lambda n: n % 3 == 0)
>>> buzz = numbers.filter(lambda n: n % 5 == 0)
>>> fizzbuzz = db.zip(fizz, buzz)
>>> list(fizzbuzzz)
[(0, 0), (3, 5), (6, 10), (9, 15), (12, 20), (15, 25), (18, 30)]
```

When what you really wanted was more along the lines of: >>> list(fizzbuzzz) # doctest: +SKIP [(0, 0), (3, None), (None, 5), (6, None), (None 10), (9, None), (12, None), (15, 15), (18, None), (None, 20), (None, 25), (None, 30)]

4.5.5 Examples

Read JSON records from disk

We commonly use `dask.bag` to process unstructured or semi-structured data:

```
>>> import dask.bag as db
>>> import json
>>> js = db.read_text('logs/2015-*.json.gz').map(json.loads)
>>> js.take(2)
({'name': 'Alice', 'location': {'city': 'LA', 'state': 'CA'}},
 {'name': 'Bob', 'location': {'city': 'NYC', 'state': 'NY'})

>>> result = js.pluck('name').frequencies()  # just another Bag
>>> dict(result)                             # Evaluate Result
{'Alice': 10000, 'Bob': 5555, 'Charlie': ...}
```

Word count

In this example, we'll use `dask` to count the number of words in text files (Enron email dataset, 6.4 GB) both locally and on a cluster (along with the distributed and hdfs3 libraries).

Local computation

Download the first text file (76 MB) in the dataset to your local machine:

```
$ wget https://s3.amazonaws.com/blaze-data/enron-email/edrm-enron-v2_allen-p_xml.zip/merged.txt
```

Import `dask.bag` and create a `bag` from the single text file:

```
>>> import dask.bag as db
>>> b = db.read_text('merged.txt', blocksize=10000000)
```

View the first ten lines of the text file with `.take()`:

```
>>> b.take(10)

('Date: Tue, 26 Sep 2000 09:26:00 -0700 (PDT)\r\n',
 'From: Phillip K Allen\r\n',
 'To: pallen70@hotmail.com\r\n',
 'Subject: Investment Structure\r\n',
 'X-SDOC: 948896\r\n',
 'X-ZLID: zl-edrm-enron-v2-allen-p-1713.eml\r\n',
 '\r\n',
 '---------------------- Forwarded by Phillip K Allen/HOU/ECT on 09/26/2000 \r\n',
 '04:26 PM --------------------------\r\n',
 '\r\n')
```

We can write a word count expression using the `bag` methods to split the lines into words, concatenate the nested lists of words into a single list, count the frequencies of each word, then list the top 10 words by their count:

```
>>> wordcount = b.str.split().concat().frequencies().topk(10, lambda x: x[1])
```

Note that the combined operations in the previous expression are lazy. We can trigger the word count computation using `.compute()`:

```
>>> wordcount.compute()

[('P', 288093),
 ('1999', 280917),
 ('2000', 277093),
 ('FO', 255844),
 ('AC', 254962),
 ('1', 240458),
 ('0', 233198),
 ('2', 224739),
 ('O', 223927),
 ('3', 221407)]
```

This computation required about 7 seconds to run on a laptop with 8 cores and 16 GB RAM.

Cluster computation with HDFS

Next, we'll use `dask` along with the distributed and hdfs3 libraries to count the number of words in all of the text files stored in a Hadoop Distributed File System (HDFS).

Copy the text data from Amazon S3 into HDFS on the cluster:

```
$ hadoop distcp s3n://AWS_SECRET_ID:AWS_SECRET_KEY@blaze-data/enron-email hdfs:///tmp/enron
```

where `AWS_SECRET_ID` and `AWS_SECRET_KEY` are valid AWS credentials.

We can now start a `distributed` scheduler and workers on the cluster, replacing `EXECUTOR_IP` and `EXECUTOR_PORT` with the IP address and port of the `distributed` scheduler:

```
$ dscheduler  # On the head node
$ dworker EXECUTOR_IP:EXECUTOR_PORT --nprocs 4 --nthreads 1  # On the compute nodes
```

Because our computations use pure Python rather than numeric libraries (e.g., NumPy, pandas), we started the workers with multiple processes rather than with multiple threads. This helps us avoid issues with the Python Global Interpreter Lock (GIL) and increases efficiency.

In Python, import the `hdfs3` and the `distributed` methods used in this example:

```
>>> import hdfs3
>>> from distributed import Executor, progress
>>> from distributed.hdfs import read_text
```

Initalize a connection to HDFS, replacing `NAMENODE_HOSTNAME` and `NAMENODE_PORT` with the hostname and port (default: 8020) of the HDFS namenode:

```
>>> hdfs = hdfs3.HDFileSystem('NAMENODE_HOSTNAME', port=NAMENODE_PORT)
```

Initalize a connection to the `distributed` executor:

```
>>> e = Executor('EXECUTOR_IP:EXECUTOR_PORT')
```

Create a `bag` from the text files stored in HDFS. This expression will not read data from HDFS until the computation is triggered:

```
>>> b = read_text('/tmp/enron/*/*', hdfs=hdfs)

Setting global dask scheduler to use distributed
```

We can write a word count expression using the same `bag` methods as the local `dask` example:

```
>>> wordcount = b.str.split().concat().frequencies().topk(10, lambda x: x[1])
```

We are ready to count the number of words in all of the text files using `distributed` workers. We can map the `wordcount` expression to a future that triggers the computation on the cluster.

```
>>> future = e.compute(wordcount)
```

Note that the `compute` operation is non-blocking, and you can continue to work in the Python shell/notebook while the computations are running.

We can check the status of the `future` while all of the text files are being processed:

```
>>> print(future)

<Future: status: pending, key: finalize-0f2f51e2350a886223f11e5a1a7bc948>
```

```
>>> progress(future)

[########################################] | 100% Completed |  8min  15.2s
```

This computation required about 8 minutes to run on a cluster with three worker machines, each with 4 cores and 16 GB RAM. For comparison, running the same computation locally with `dask` required about 20 minutes on a single machine with the same specs.

When the `future` finishes reading in all of the text files and counting words, the results will exist on each worker. To sum the word counts for all of the text files, we need to gather the results from the `distributed` workers:

```
>>> results = e.gather(future)
```

Finally, we print the top 10 words from all of the text files:

```
>>> print(results)

[('0', 67218227),
 ('the', 19588747),
 ('-', 14126955),
 ('to', 11893912),
 ('N/A', 11814994),
 ('of', 11725144),
 ('and', 10254267),
 ('in', 6685245),
 ('a', 5470711),
 ('or', 5227787)]
```

The complete Python script for this example is shown below:

```
# word-count.py

# Local computation

import dask.bag as db
b = db.read_text('merged.txt')
b.take(10)
wordcount = b.str.split().concat().frequencies().topk(10, lambda x: x[1])
wordcount.compute()

# Cluster computation with HDFS

import hdfs3
from distributed import Executor, progress
from distributed.hdfs import read_text

hdfs = hdfs3.HDFileSystem('NAMENODE_HOSTNAME', port=NAMENODE_PORT)
e = Executor('EXECUTOR_IP:EXECUTOR_PORT')

b = read_text('/tmp/enron/*/*', hdfs=hdfs)
wordcount = b.str.split().concat().frequencies().topk(10, lambda x: x[1])

future = e.compute(wordcount)
print(future)
progress(future)

results = e.gather(future)
print(results)
```

4.6 DataFrame

Dask dataframes look and feel like pandas dataframes, but operate on datasets larger than memory using multiple threads. Dask.dataframe does not implement the complete pandas interface.

4.6.1 Overview

Dask Dataframe implements a subset of the Pandas Dataframe interface using blocked algorithms, cutting up the large DataFrame into many small Pandas DataFrames. This lets us compute on dataframes that are larger than memory using all of our cores. We coordinate these blocked algorithms using dask graphs.

Design

Dask dataframes coordinate many Pandas arrays arranged along an index. These Pandas dataframes may live on disk or on other machines.

Common Uses

Dask dataframes are commonly used to quickly inspect and analyze large volumes of tabular data stored in CSV, HDF5, or other tabular formats. This includes experimental data, financial data, or any other case where you have many heterogeneously typed columns that don't fit into memory.

Dask.dataframe copies the pandas API

Because the `dask.dataframe` application programming interface (API) is a subset of the pandas API it should be familiar to pandas users. There are some slight alterations due to the parallel nature of dask:

```
>>> import dask.dataframe as dd
>>> df = dd.read_csv('2014-*.csv')
>>> df.head()
   x  y
0  1  a
1  2  b
2  3  c
3  4  a
4  5  b
5  6  c

>>> df2 = df[df.y == 'a'].x + 1
```

As with all dask collections (for example Array, Bag, DataFrame) one triggers computation by calling the `.compute()` method:

```
>>> df2.compute()
0    2
3    5
Name: x, dtype: int64
```

Scope

Dask.dataframe covers a small but well-used portion of the pandas API. This limitation is for two reasons:

1. The pandas API is *huge*
2. Some operations are genuinely hard to do in parallel (for example sort).

Additionally, some important operations like `set_index` work, but are slower than in pandas because they may write out to disk.

The following class of computations works well:

- **Trivially parallelizable operations (fast):**
  - Elementwise operations: `df.x + df.y`, `df * df`
  - Row-wise selections: `df[df.x > 0]`
  - Loc: `df.loc[4.0:10.5]`
  - Common aggregations: `df.x.max()`, `df.max()`
  - Is in: `df[df.x.isin([1, 2, 3])]`
  - Datetime/string accessors: `df.timestamp.month`
- **Cleverly parallelizable operations (fast):**
  - groupby-aggregate (with common aggregations): `df.groupby(df.x).y.max()`, `df.groupby('x').max()`
  - value_counts: `df.x.value_counts()`
  - Drop duplicates: `df.x.drop_duplicates()`
  - Join on index: `dd.merge(df1, df2, left_index=True, right_index=True)`
  - Join with Pandas DataFrames: `dd.merge(df1, df2, on='id')`
  - Elementwise operations with different partitions / divisions: `df1.x + df2.y`
  - Datetime resampling: `df.resample(...)`
  - Rolling averages: `df.rolling(...)`
  - Pearson Correlations: `df[['col1', 'col2']].corr()`
- **Operations requiring a shuffle (slow-ish, unless on index)**
  - Set index: `df.set_index(df.x)`
  - groupby-apply (with anything): `df.groupby(df.x).apply(myfunc)`
  - Join not on the index: `dd.merge(df1, df2, on='name')`

See DataFrame API documentation for a more extensive list.

Execution

By default `dask.dataframe` uses the multi-threaded scheduler. This exposes some parallelism when pandas or the underlying numpy operations release the global interpreter lock (GIL). Generally pandas is more GIL bound than NumPy, so multi-core speed-ups are not as pronounced for `dask.dataframe` as they are for `dask.array`. This is changing, and the pandas development team is actively working on releasing the GIL.

In some cases you may experience speedups by switching to the multiprocessing or distributed scheduler.

```
>>> dask.set_options(get=dask.multiprocessing.get)
```

See scheduler docs for more information.

Limitations

Dask.DataFrame does not implement the entire Pandas interface. Users expecting this will be disappointed. Notably, dask.dataframe has the following limitations:

1. Setting a new index from an unsorted column is expensive
2. Many operations, like groupby-apply and join on unsorted columns require setting the index, which as mentioned above, is expensive
3. The Pandas API is very large. Dask.dataframe does not attempt to implement many pandas features or any of the more exotic data structures like NDFrames

4.6.2 Create Dask DataFrames

From CSV files

`dask.dataframe.read_csv` uses `pandas.read_csv` and so inherits most of that function's options. Additionally it gains two new functionalities

1. You can provide a globstring

```
>>> df = dd.read_csv('data.*.csv')
```

2. You can specify the size of each block of data in bytes of uncompressed data. Note that, especially for text data the size on disk may be much less than the number of bytes in memory.

```
>>> df = dd.read_csv('data.csv', chunkbytes=1000000)  # 1MB chunks
```

From an Array

You can create a DataFrame from any sliceable array like object including both NumPy arrays and HDF5 datasets.

```
>>> dd.from_array(x, chunksize=1000000)
```

From BColz

BColz is an on-disk, chunked, compressed, column-store. These attributes make it very attractive for dask.dataframe which can operate particularly well on it. There is a special `from_bcolz` function.

```
>>> df = dd.from_bcolz('myfile.bcolz', chunksize=1000000)
```

In particular column access on a dask.dataframe backed by a `bcolz.ctable` will only read the necessary columns from disk. This can provide dramatic performance improvements.

From Castra

Castra is a tiny, experimental partitioned on-disk data structure designed to fit the `dask.dataframe` model. It provides columnstore access and range queries along the index. It is also a very small project (roughly 400 lines) and not actively maintained; use at your own risk.

```
>>> from castra import Castra
>>> c = Castra(path='/my/castra/file')
>>> df = c.to_dask()
```

From Bags

You can create a dask.dataframe from a dask bag.

dask.bag.core.Bag.to_dataframe([columns])	Convert Bag to dask.dataframe

Using `dask.delayed`

You can create a plan to arrange many Pandas frames into a sequence with normal for loops using dask.delayed and then convert these into a dask dataframe later. See documentation on using dask.delayed with collections or an example notebook. Dask.delayed is more useful when simple `map` operations aren't sufficient to capture the complexity of your data layout.

From Raw Dask Graphs

This section is for developer information and discusses internal API. You should never need to create a dataframe object by hand.

To construct a DataFrame manually from a dask graph you need the following information:

1. dask: a dask graph with keys like `{(name, 0): ..., (name, 1): ...}` as well as any other tasks on which those tasks depend. The tasks corresponding to `(name, i)` should produce `pandas.DataFrame` objects that correspond to the columns and divisions information discussed below.
2. name: The special name used above
3. columns: A list of column names
4. divisions: A list of index values that separate the different partitions. Alternatively, if you don't know the divisions (this is common) you can provide a list of `[None, None, None, ...]` with as many partitions as you have plus one. For more information see the Partitions section in the dataframe documentation.

As an example, we build a DataFrame manually that reads several CSV files that have a datetime index separated by day. Note, you should never do this. The `dd.read_csv` function does this for you.

```
dsk = {('mydf', 0): (pd.read_csv, 'data/2000-01-01.csv'),
       ('mydf', 1): (pd.read_csv, 'data/2000-01-02.csv'),
       ('mydf', 2): (pd.read_csv, 'data/2000-01-03.csv')}
name = 'mydf'
columns = ['price', 'name', 'id']
divisions = [Timestamp('2000-01-01 00:00:00'),
             Timestamp('2000-01-02 00:00:00'),
             Timestamp('2000-01-03 00:00:00'),
             Timestamp('2000-01-03 23:59:59')]

df = dd.DataFrame(dsk, name, columns, divisions)
```

4.6.3 API

Top level user functions:

DataFrame	Implements out-of-core DataFrame as a sequence of pandas DataFrames
DataFrame.add(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *add*).
DataFrame.append(other)	Append rows of *other* to the end of this frame, returning a new object.
DataFrame.apply(func[, axis, args, columns])	Parallel version of pandas.DataFrame.apply
DataFrame.assign(**kwargs)	Assign new columns to a DataFrame, returning a new object (a copy) with
DataFrame.astype(dtype)	Cast object to input numpy.dtype
DataFrame.cache([cache])	Evaluate Dataframe and store in local cache
DataFrame.categorize([columns])	Convert columns of the DataFrame to catefory dtype
DataFrame.column_info	Return DataFrame.columns
DataFrame.columns	
DataFrame.compute(**kwargs)	Compute several dask collections at once.
DataFrame.corr([method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values
DataFrame.count([axis])	Return Series with number of non-NA/null observations over requested ax
DataFrame.cov([min_periods])	Compute pairwise covariance of columns, excluding NA/null values
DataFrame.cummax([axis, skipna])	Return cumulative cummax over requested axis.
DataFrame.cummin([axis, skipna])	Return cumulative cummin over requested axis.
DataFrame.cumprod([axis, skipna])	Return cumulative cumprod over requested axis.
DataFrame.cumsum([axis, skipna])	Return cumulative cumsum over requested axis.
DataFrame.describe()	Generate various summary statistics, excluding NaN values.
DataFrame.div(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *tr*
DataFrame.drop(labels[, axis])	Return new object with labels in requested axis removed.
DataFrame.drop_duplicates(**kwargs)	Return DataFrame with duplicate rows removed, optionally only
DataFrame.dropna([how, subset])	Return object with labels on given axis omitted where alternately any
DataFrame.dtypes	Return data types
DataFrame.fillna(value)	Fill NA/NaN values using the specified method
DataFrame.floordiv(other[, axis, level, ...])	Integer division of dataframe and other, element-wise (binary operator *floo*
DataFrame.get_division(n)	Get nth division of the data
DataFrame.groupby(key, **kwargs)	Group series using mapper (dict or key function, apply given function to g
DataFrame.head([n, compute])	First n rows of the dataset
DataFrame.iloc	
DataFrame.index	Return dask Index instance
DataFrame.iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
DataFrame.itertuples()	Iterate over DataFrame rows as namedtuples, with index value as first elen
DataFrame.join(other[, on, how, lsuffix, ...])	Join columns with other DataFrame either on index or on a key column.
DataFrame.known_divisions	Whether divisions are already known
DataFrame.loc	Purely label-location based indexer for selection by label.
DataFrame.map_partitions(func[, columns])	Apply Python function on each DataFrame block
DataFrame.mask(cond[, other])	Return an object of same shape as self and whose corresponding entries ar
DataFrame.max([axis, skipna])	This method returns the maximum of the values in the object.
DataFrame.mean([axis, skipna])	Return the mean of the values for the requested axis
DataFrame.merge(right[, how, on, left_on, ...])	Merge DataFrame objects by performing a database-style join operation b
DataFrame.min([axis, skipna])	This method returns the minimum of the values in the object.
DataFrame.mod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *mod*).
DataFrame.mul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*)
DataFrame.ndim	Return dimensionality
DataFrame.nlargest([n, columns])	Get the rows of a DataFrame sorted by the *n* largest values of *columns*.
DataFrame.npartitions	Return number of partitions

Table 4.11 – continued from previous pag

DataFrame.pow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator
DataFrame.quantile([q, axis])	Approximate row-wise and precise column-wise quantiles of DataFrame
DataFrame.query(expr, **kwargs)	
DataFrame.radd(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *radd*).
DataFrame.random_split(p[, random_state])	Pseudorandomly split dataframe into different pieces row-wise
DataFrame.rdiv(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *rt*
DataFrame.rename([index, columns])	Alter axes input function or functions.
DataFrame.repartition([divisions, ...])	Repartition dataframe along new divisions
DataFrame.reset_index()	For DataFrame with multi-level index, return new DataFrame with labelin
DataFrame.rfloordiv(other[, axis, level, ...])	Integer division of dataframe and other, element-wise (binary operator *rflo*
DataFrame.rmod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *rmod*).
DataFrame.rmul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *rmul*
DataFrame.rpow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator
DataFrame.rsub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *rsub*).
DataFrame.rtruediv(other[, axis, level, ...])	Floating division of dataframe and other, element-wise (binary operator *rt*
DataFrame.sample(frac[, replace, random_state])	Random sample of items
DataFrame.set_index(other[, drop, sorted])	Set the DataFrame index (row labels) using an existing column
DataFrame.set_partition(column, divisions, ...)	Set explicit divisions for new column index
DataFrame.std([axis, skipna, ddof])	Return sample standard deviation over requested axis.
DataFrame.sub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
DataFrame.sum([axis, skipna])	Return the sum of the values for the requested axis
DataFrame.tail([n, compute])	Last n rows of the dataset
DataFrame.to_bag([index])	Convert to a dask Bag of tuples of each row.
DataFrame.to_castra([fn, categories, ...])	Write DataFrame to Castra on-disk store
DataFrame.to_csv(filename[, get])	Write DataFrame to a comma-separated values (csv) file
DataFrame.to_hdf(path_or_buf, key[, mode, ...])	Activate the HDFStore.
DataFrame.to_delayed()	Convert dataframe into dask Values
DataFrame.truediv(other[, axis, level, ...])	Floating division of dataframe and other, element-wise (binary operator *tr*
DataFrame.var([axis, skipna, ddof])	Return unbiased variance over requested axis.
DataFrame.visualize([filename, format, ...])	Render the computation of this object's task graph using graphviz.
DataFrame.where(cond[, other])	Return an object of same shape as self and whose corresponding entries ar

Rolling Operations

rolling.rolling_apply(arg, window, *args, ...)	Generic moving function application.
rolling.rolling_chunk(func, part1, part2, ...)	
rolling.rolling_count(arg, window, *args, ...)	Rolling count of number of non-NaN observations inside provided win
rolling.rolling_kurt(arg, window, *args, ...)	Unbiased moving kurtosis.
rolling.rolling_max(arg, window, *args, **kwargs)	Moving maximum.
rolling.rolling_mean(arg, window, *args, ...)	Moving mean.
rolling.rolling_median(arg, window, *args, ...)	Moving median.
rolling.rolling_min(arg, window, *args, **kwargs)	Moving minimum.
rolling.rolling_quantile(arg, window, *args, ...)	Moving quantile.
rolling.rolling_skew(arg, window, *args, ...)	Unbiased moving skewness.
rolling.rolling_std(arg, window, *args, **kwargs)	Moving standard deviation.
rolling.rolling_sum(arg, window, *args, **kwargs)	Moving sum.
rolling.rolling_var(arg, window, *args, **kwargs)	Moving variance.
rolling.rolling_window(arg, window, *args, ...)	Applies a moving window of type window_type and size window c

Create DataFrames

from_array(x[, chunksize, columns])	Read dask Dataframe from any slicable array
from_bcolz(x[, chunksize, categorize, ...])	Read dask Dataframe from bcolz.ctable
from_castra(x[, columns])	Load a dask DataFrame from a Castra.
read_csv(urlpath[, blocksize, chunkbytes, ...])	Read CSV files into a Dask.DataFrame
from_dask_array(x[, columns])	Convert dask Array to dask DataFrame
from_delayed(dfs[, metadata, divisions, ...])	Create DataFrame from many dask.delayed objects
from_pandas(data[, npartitions, chunksize, ...])	Construct a dask object from a pandas object.

DataFrame Methods

class dask.dataframe.**DataFrame**

Implements out-of-core DataFrame as a sequence of pandas DataFrames

Parameters **dask: dict**

The dask graph to compute this DataFrame

name: str

The key prefix that specifies which keys in the dask comprise this particular DataFrame

columns: list of str

Column names. This metadata aids usability

divisions: tuple of index values

Values along which we partition our blocks on the index

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to dataframe + other, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.radd

Notes

Mismatched indices will be unioned together

append(*other*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

Parameters **other** : DataFrame or Series/dict-like object, or list of these

The data to append.

ignore_index : boolean, default False

If True, do not use the index labels.

verify_integrity : boolean, default False

If True, raise ValueError on creating index with duplicates.

Returns **appended** : DataFrame

See also:

pandas.concat General function to concatenate DataFrame, Series or Panel objects

Notes

Dask doesn't supports following argument(s).

- ignore_index
- verify_integrity

Examples

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

apply(*func*, *axis=0*, *args=()*, *columns='__no_default__'*, ***kwds*)

Parallel version of pandas.DataFrame.apply

This mimics the pandas version except for the following:

1. The user must specify axis=1 explicitly.

2.The user should provide output columns.

Parameters **func: function**

Function to apply to each column

axis: {0 or 'index', 1 or 'columns'}, default 0

- 0 or 'index': apply function to each column (NOT SUPPORTED)
- 1 or 'columns': apply function to each row

columns: list, scalar or None

If list is given, the result is a DataFrame which columns is specified list. Otherwise, the result is a Series which name is given scalar or None (no name). If name keyword is not given, dask tries to infer the result type using its beginning of data. This inference may take some time and lead to unexpected result

args : tuple

Positional arguments to pass to function in addition to the array/series

Additional keyword arguments will be passed as keywords to the function

Returns **applied** : Series or DataFrame depending on name keyword

assign (***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with all the original columns in addition to the new ones.

New in version 0.16.0.

Parameters **kwargs** : keyword, value pairs

keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

Returns **df** : DataFrame

A new DataFrame with the new columns in addition to all the existing columns.

Notes

Since `kwargs` is a dictionary, the order of your arguments may not be preserved. The make things predicatable, the columns are inserted in alphabetical order, at the end of your DataFrame. Assigning multiple columns within the same `assign` is possible, but you cannot reference other columns created within the same `assign` call.

Examples

```
>>> df = DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

astype(*dtype*)

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters **dtype** : numpy.dtype or Python type

raise_on_error : raise on invalid input

kwargs : keyword arguments to pass on to the constructor

Returns **casted** : type of caller

Notes

Dask doesn't supports following argument(s).

- copy
- raise_on_error

cache(*cache=<type 'dict'>*)

Evaluate Dataframe and store in local cache

Uses chest by default to store data on disk

categorize(*columns=None*, ***kwargs*)

Convert columns of the DataFrame to catefory dtype

Parameters **columns** : list, optional

A list of column names to convert to the category type. By default any column with an object dtype is converted to a categorical.

kwargs

Keyword arguments are passed on to compute.

See also:

`dask.dataframes.categorical.categorize`

Notes

When dealing with columns of repeated text values converting to categorical type is often much more performant, both in terms of memory and in writing to disk or communication over the network.

column_info
Return DataFrame.columns

compute(***kwargs*)
Compute several dask collections at once.

Parameters **get** : callable, optional

A scheduler `get` function to use. If not provided, the default is to check the global settings first, and then fall back to the collection defaults.

optimize_graph : bool, optional

If True [default], the graph is optimized before computation. Otherwise the graph is run as is. This can be useful for debugging.

kwargs

Extra keywords to forward to the scheduler `get` function.

corr(*method='pearson'*, *min_periods=None*)
Compute pairwise correlation of columns, excluding NA/null values

Parameters **method** : {'pearson', 'kendall', 'spearman'}

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods : int, optional

Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

Returns **y** : DataFrame

count(*axis=None*)
Return Series with number of non-NA/null observations over requested axis. Works with non-floating point data as well (detects NaN and None)

Parameters **axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only : boolean, default False

Include only float, int, boolean data

Returns **count** : Series (or DataFrame if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

cov (*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values

Parameters **min_periods** : int, optional

Minimum number of observations required per pair of columns to have a valid result.

Returns **y** : DataFrame

Notes

y contains the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1 (unbiased estimator).

cummax (*axis=None*, *skipna=True*)

Return cumulative cummax over requested axis.

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummax** : Series

Notes

Dask doesn't supports following argument(s).

- dtype
- out

cummin (*axis=None*, *skipna=True*)

Return cumulative cummin over requested axis.

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummin** : Series

Notes

Dask doesn't supports following argument(s).

- dtype

- out

cumprod(*axis=None*, *skipna=True*)

Return cumulative cumprod over requested axis.

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumprod** : Series

Notes

Dask doesn't supports following argument(s).

- dtype
- out

cumsum(*axis=None*, *skipna=True*)

Return cumulative cumsum over requested axis.

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumsum** : Series

Notes

Dask doesn't supports following argument(s).

- dtype
- out

describe()

Generate various summary statistics, excluding NaN values.

Parameters **percentiles** : array-like, optional

The percentiles to include in the output. Should all be in the interval [0, 1]. By default *percentiles* is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.

include, exclude : list-like, 'all', or None (default)

Specify the form of the returned result. Either:

- None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.
- A list of dtypes or strings to be included/excluded. To select all numeric types use numpy numpy.number. To select categorical objects use type object. See also the select_dtypes documentation. eg. df.describe(include=['O'])
- If include is the string 'all', the output column-set will match the input one.

Returns summary: NDFrame of summary statistics

See also:

`DataFrame.select_dtypes`

Notes

Dask doesn't supports following argument(s).

- percentiles
- include
- exclude

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.rtruediv

Notes

Mismatched indices will be unioned together

drop(*labels*, *axis=0*)
Return new object with labels in requested axis removed.

Parameters **labels** : single label or list-like

axis : int or axis name

level : int or level name, default None

For MultiIndex

inplace : bool, default False

If True, do operation inplace and return None.

errors : {'ignore', 'raise'}, default 'raise'

If 'ignore', suppress error and existing labels are dropped.

New in version 0.16.1.

Returns **dropped** : type of caller

Notes

Dask doesn't supports following argument(s).

- level
- inplace
- errors

drop_duplicates(***kwargs*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

Parameters **subset** : column label or sequence of labels, optional

Only consider certain columns for identifying duplicates, by default use all of the columns

keep : {'first', 'last', False}, default 'first'

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

take_last : deprecated

inplace : boolean, default False

Whether to drop duplicates in place or to return a copy

Returns **deduplicated** : DataFrame

dropna(*how='any'*, *subset=None*)

Return object with labels on given axis omitted where alternately any or all of the data are missing

Parameters **axis** : {0 or 'index', 1 or 'columns'}, or tuple/list thereof

Pass tuple or list to drop on multiple axes

how : {'any', 'all'}

- any : if any NA values are present, drop that label
- all : if all values are NA, drop that label

thresh : int, default None

int value : require that many non-NA values

subset : array-like

Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include

inplace : boolean, default False

If True, do operation inplace and return None.

Returns **dropped** : DataFrame

Notes

Dask doesn't supports following argument(s).

- axis
- thresh
- inplace

dtypes
Return data types

eval(*expr*, *inplace=None*, ***kwargs*)
Evaluate an expression in the context of the calling DataFrame instance.

Parameters **expr** : string

The expression string to evaluate.

inplace : bool

If the expression contains an assignment, whether to return a new DataFrame or mutate the existing.

WARNING: inplace=None currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

New in version 0.18.0.

kwargs : dict

See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

Returns **ret** : ndarray, scalar, or pandas object

See also:

`pandas.DataFrame.query`, `pandas.DataFrame.assign`, `pandas.eval`

Notes

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

Examples

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = DataFrame(randn(10, 2), columns=list('ab'))
>>> df.eval('a + b')
>>> df.eval('c = a + b')
```

fillna(*value*)
Fill NA/NaN values using the specified method

Parameters **value** : scalar, dict, Series, or DataFrame

Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0, 'index'}

inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit : int, default None

If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

downcast : dict, default is None

a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

Returns filled : Series

See also:

`reindex`, `asfreq`

Notes

Dask doesn't supports following argument(s).

- method
- axis
- inplace
- limit
- downcast

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters other : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.rfloordiv

Notes

Mismatched indices will be unioned together

get_division(*n*)
Get nth division of the data

groupby(*key*, ***kwargs*)
Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

Parameters **by** : mapping function / list of functions, dict, Series, or tuple /

list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups

axis : int, default 0

level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort : boolean, default True

Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys : boolean, default True

When calling apply, add group keys to index to identify pieces

squeeze : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

Returns GroupBy object

Notes

Dask doesn't supports following argument(s).

- by
- axis
- level

- as_index
- sort
- group_keys
- squeeze

Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

head(*n=5*, *compute=True*)
First n rows of the dataset

Caveat, this only checks the first n rows of the first partition.

index
Return dask Index instance

info()
Concise summary of a Dask DataFrame.

isnull()
Return a boolean same-sized object indicating if the values are null.

See also:

notnull boolean inverse of isnull

iterrows()
Iterate over DataFrame rows as (index, Series) pairs.

Returns it : generator

A generator that iterates over the rows of the frame.

See also:

itertuples Iterate over DataFrame rows as namedtuples of the values.

iteritems Iterate over (column name, Series) pairs.

Notes

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
```

```
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

itertuples()

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

Parameters **index** : boolean, default True

If True, return the index as the first element of the tuple.

name : string, default "Pandas"

The name of the returned namedtuples or None to return regular tuples.

See also:

iterrows Iterate over DataFrame rows as (index, Series) pairs.

iteritems Iterate over (column name, Series) pairs.

Notes

Dask doesn't supports following argument(s).

- index
- name

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

join(*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *npartitions=None*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

Parameters **other** : DataFrame, Series with name field set, or list of DataFrame

Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on : column name, tuple/list of column names, or array-like

Column(s) to use for joining, otherwise join on index. If multiples columns given, the passed DataFrame must have a MultiIndex. Can pass an array as the join key if not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how : {'left', 'right', 'outer', 'inner'}

How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise

- left: use calling frame's index
- right: use input frame's index
- outer: form union of indexes
- inner: use intersection of indexes

lsuffix : string

Suffix to use from left frame's overlapping columns

rsuffix : string

Suffix to use from right frame's overlapping columns

sort : boolean, default False

Order result DataFrame lexicographically by the join key. If False, preserves the index order of the calling (left) DataFrame

Returns **joined** : DataFrame

Notes

Dask doesn't supports following argument(s).

- sort

known_divisions
Whether divisions are already known

loc
Purely label-location based indexer for selection by label.

```
>>> df.loc["b"]
>>> df.loc["b":"d"]
```

map_partitions (*func*, *columns='__no_default__'*, **args*, ***kwargs*)
Apply Python function on each DataFrame block

When using `map_partitions` you should provide either the column names (if the result is a DataFrame) or the name of the Series (if the result is a Series). The output type will be determined by the type of `columns`.

Parameters **func** : function

Function applied to each blocks

columns : tuple or scalar

Column names or name of the output. Defaults to names of data itself. When tuple is passed, DataFrame is returned. When scalar is passed, Series is returned.

Examples

When str is passed as columns, the result will be Series.

```
>>> df.map_partitions(lambda df: df.x + 1, columns='x')
```

When tuple is passed as columns, the result will be Series.

```
>>> df.map_partitions(lambda df: df.head(), columns=df.columns)
```

mask(*cond*, *other=nan*)

Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

Parameters **cond** : boolean NDFrame, array or callable

If cond is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as cond.

other : scalar, NDFrame, or callable

If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

try_cast : boolean, default False

try to cast the result back to the input type (if possible),

raise_on_error : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Returns **wh** : same type as caller

Notes

Dask doesn't supports following argument(s).

- inplace
- axis

- level
- try_cast
- raise_on_error

max(*axis=None*, *skipna=True*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **max** : Series or DataFrame (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

mean(*axis=None*, *skipna=True*)

Return the mean of the values for the requested axis

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **mean** : Series or DataFrame (if level specified)

Notes

Dask doesn't supports following argument(s).

- level

•numeric_only

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *suffixes=('_x', '_y')*, *npartitions=None*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters **right** : DataFrame

how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join)

on : label or list

Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

left_on : label or list, or array-like

Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns

right_on : label or list, or array-like

Field names to join on in right DataFrame or vector/list of vectors per left_on docs

left_index : boolean, default False

Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index : boolean, default False

Use the index from the right DataFrame as the join key. Same caveats as left_index

sort : boolean, default False

Sort the join keys lexicographically in the result DataFrame

suffixes : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

copy : boolean, default True

If False, do not copy data unnecessarily

indicator : boolean or string, default False

If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

New in version 0.17.0.

Returns **merged** : DataFrame

The output type will the be same as 'left', if it is a subclass of DataFrame.

Notes

Dask doesn't supports following argument(s).

- sort
- copy
- indicator

Examples

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

min(*axis=None*, *skipna=True*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **min** : Series or DataFrame (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.rmod

Notes

Mismatched indices will be unioned together

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.rmul

Notes

Mismatched indices will be unioned together

ndim
Return dimensionality

nlargest (*n=5*, *columns=None*)
Get the rows of a DataFrame sorted by the *n* largest values of *columns*.

New in version 0.17.0.

Parameters **n** : int

Number of items to retrieve

columns : list or str

Column name or names to order by

keep : {'first', 'last', False}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns DataFrame

Notes

Dask doesn't supports following argument(s).

•keep

Examples

```
>>> df = DataFrame({'a': [1, 10, 8, 11, -1],
...                 'b': list('abdce'),
...                 'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nlargest(3, 'a')
    a  b   c
3  11  c   3
1  10  b   2
2   8  d NaN
```

notnull ()
Return a boolean same-sized object indicating if the values are not null.

See also:

isnull boolean inverse of notnull

npartitions
Return number of partitions

pow (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.rpow

Notes

Mismatched indices will be unioned together

quantile(*q=0.5*, *axis=0*)

Approximate row-wise and precise column-wise quantiles of DataFrame

Parameters **q** : list/array of floats, default 0.5 (50%)

Iterable of numbers ranging from 0 to 1 for the desired quantiles

axis : {0, 1, 'index', 'columns'} (default 0)

0 or 'index' for row-wise, 1 or 'columns' for column-wise

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.add

Notes

Mismatched indices will be unioned together

random_split(*p*, *random_state=None*)
Pseudorandomly split dataframe into different pieces row-wise

Parameters **frac** : float, optional

Fraction of axis items to return.

random_state: int or np.random.RandomState

If int create a new RandomState with this as the seed

Otherwise draw from the passed RandomState

Examples

50/50 split

```
>>> a, b = df.random_split([0.5, 0.5])
```

80/10/10 split, consistent random_state

```
>>> a, b, c = df.random_split([0.8, 0.1, 0.1], random_state=123)
```

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.truediv

Notes

Mismatched indices will be unioned together

rename(*index=None*, *columns=None*)
Alter axes input function or functions. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Alternatively, change `Series.name` with a scalar value (Series only).

Parameters **index, columns** : scalar, list-like, dict-like or function, optional

Scalar or list-like will alter the `Series.name` attribute, and raise on DataFrame or Panel. dict-like or functions are transformations to apply to that axis' values

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Whether to return a new DataFrame. If True then value of copy is ignored.

Returns **renamed** : DataFrame (new object)

See also:

`pandas.NDFrame.rename_axis`

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
3    2
5    3
dtype: int64
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
...
TypeError: 'int' object is not callable
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

repartition(*divisions=None*, *npartitions=None*, *force=False*)
Repartition dataframe along new divisions

Parameters **divisions** : list

List of partitions to be used

npartitions : int

Number of partitions of output, must be less than npartitions of input

force : bool, default False

Allows the expansion of the existing divisions. If False then the new divisions lower and upper bounds must be the same as the old divisions.

Examples

```
>>> df = df.repartition(npartitions=10)
>>> df = df.repartition(divisions=[0, 5, 10, 20])
```

reset_index()

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

Parameters **level** : int, str, tuple, or list, default None

Only remove the given levels from the index. Removes all levels by default

drop : boolean, default False

Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace : boolean, default False

Modify the DataFrame in place (do not create a new object)

col_level : int or str, default 0

If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill : object, default ''

If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

Returns **resetted** : DataFrame

Notes

Dask doesn't supports following argument(s).

- level
- drop
- inplace
- col_level
- col_fill

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.floordiv

Notes

Mismatched indices will be unioned together

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.mod

Notes

Mismatched indices will be unioned together

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.mul

Notes

Mismatched indices will be unioned together

rolling(*window*, *min_periods=None*, *freq=None*, *center=False*, *win_type=None*, *axis=0*)
Provides rolling transformations.

Parameters **window** : int

Size of the moving window. This is the number of observations used for calculating the statistic. The window size must not be so large as to span more than one adjacent partition.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

center : boolean, default False

Set the labels at the center of the window.

win_type : string, default None

Provide a window type. The recognized window types are identical to pandas.

axis : int, default 0

Returns a Rolling object on which to call a method to compute a statistic

Notes

The *freq* argument is not supported.

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.pow

Notes

Mismatched indices will be unioned together

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.sub

Notes

Mismatched indices will be unioned together

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : DataFrame

See also:

DataFrame.truediv

Notes

Mismatched indices will be unioned together

sample(*frac*, *replace=False*, *random_state=None*)
Random sample of items

Parameters frac : float, optional

Fraction of axis items to return.

replace: boolean, optional

Sample with or without replacement. Default = False.

random_state: int or ‘‘np.random.RandomState‘‘

If int we create a new RandomState with this as the seed Otherwise we draw from the passed RandomState

set_index(*other*, *drop=True*, *sorted=False*, ***kwargs*)
Set the DataFrame index (row labels) using an existing column

This operation in dask.dataframe is expensive. If the input column is sorted then we accomplish the set_index in a single full read of that column. However, if the input column is not sorted then this operation triggers a full shuffle, which can take a while and only works on a single machine (not distributed).

Parameters other: Series or label

drop: boolean, default True

Delete columns to be used as the new index

sorted: boolean, default False

Set to True if the new index column is already sorted

Examples

```
>>> df.set_index('x')
>>> df.set_index(d.x)
>>> df.set_index(d.timestamp, sorted=True)
```

set_partition(*column*, *divisions*, ***kwargs*)
Set explicit divisions for new column index

```
>>> df2 = df.set_partition('new-index-column', divisions=[10, 20, 50])
```

See also:

set_index

std(*axis=None*, *skipna=True*, *ddof=1*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof : int, default 1

degrees of freedom

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **std** : Series or DataFrame (if level specified)

Notes

Dask doesn't supports following argument(s).

•level

•numeric_only

sub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.rsub

Notes

Mismatched indices will be unioned together

sum(*axis=None*, *skipna=True*)
Return the sum of the values for the requested axis

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **sum** : Series or DataFrame (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

tail(*n=5*, *compute=True*)
Last n rows of the dataset

Caveat, the only checks the last n rows of the last partition.

to_bag(*index=False*)
Convert to a dask Bag of tuples of each row.

Parameters **index** : bool, optional

If True, the index is included as the first element of each tuple. Default is False.

to_castra(*fn=None*, *categories=None*, *sorted_index_column=None*, *compute=True*, *get=<function get_sync>*)
Write DataFrame to Castra on-disk store

See https://github.com/blosc/castra for details

See also:

`Castra.to_dask`

to_csv(*filename*, *get=<function get_sync>*, ***kwargs*)
Write DataFrame to a comma-separated values (csv) file

Parameters **path_or_buf** : string or file handle, default None

File path or object, if None is provided the result is returned as a string.

sep : character, default ','

Field delimiter for the output file.

na_rep : string, default ‘’

Missing data representation

float_format : string, default None

Format string for floating point numbers

columns : sequence, optional

Columns to write

header : boolean or list of string, default True

Write out column names. If a list of string is given it is assumed to be aliases for the column names

index : boolean, default True

Write row names (index)

index_label : string or sequence, or False, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

nanRep : None

deprecated, use na_rep

mode : str

Python write mode, default ‘w’

encoding : string, optional

A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.

compression : string, optional

a string representing the compression to use in the output file, allowed values are ‘gzip’, ‘bz2’, ‘xz’, only used when the first argument is a filename

line_terminator : string, default ‘n’

The newline character or character sequence to use in the output file

quoting : optional constant from csv module

defaults to csv.QUOTE_MINIMAL

quotechar : string (length 1), default ‘”’

character used to quote fields

doublequote : boolean, default True

Control quoting of *quotechar* inside a field

escapechar : string (length 1), default None

character used to escape *sep* and *quotechar* when appropriate

chunksize : int or None

rows to write at a time

tupleize_cols : boolean, default False

write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

date_format : string, default None

Format string for datetime objects

decimal: string, default '.'

Character recognized as decimal separator. E.g. use ',' for European data

New in version 0.16.0.

Notes

Dask doesn't supports following argument(s).

- •path_or_buf
- •sep
- •na_rep
- •float_format
- •columns
- •header
- •index
- •index_label
- •mode
- •encoding
- •compression
- •quoting
- •quotechar
- •line_terminator
- •chunksize
- •tupleize_cols
- •date_format
- •doublequote
- •escapechar
- •decimal

to_delayed()

Convert dataframe into dask Values

Returns a list of values, one value per partition.

to_hdf(*path_or_buf*, *key*, *mode='a'*, *append=False*, *complevel=0*, *complib=None*, *fletcher32=False*, *get=<function get_sync>*, ***kwargs*)

Activate the HDFStore.

Parameters **path_or_buf** : the path (string) or HDFStore object

key : string

indentifier for the group in the store

mode : optional, {'a', 'w', 'r', 'r+'}, default 'a'

'r' Read-only; no data can be modified.

'w' Write; a new file is created (an existing file with the same name would be deleted).

'a' Append; an existing file is opened for reading and writing, and if the file does not exist it is created.

'r+' It is similar to 'a', but the file must already exist.

format : 'fixed(f)|table(t)', default is 'fixed'

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable

table(t) [Table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data

append : boolean, default False

For Table formats, append the input data to the existing

complevel : int, 1-9, default 0

If a complib is specified compression will be applied where possible

complib : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dropna : boolean, default False.

If true, ALL nan rows will not be written to store.

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

DataFrame.rtruediv

Notes

Mismatched indices will be unioned together

var (*axis=None*, *skipna=True*, *ddof=1*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters **axis** : {index (0), columns (1)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof : int, default 1

degrees of freedom

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **var** : Series or DataFrame (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

visualize (*filename='mydask'*, *format=None*, *optimize_graph=False*, ***kwargs*)
Render the computation of this object's task graph using graphviz.

Requires `graphviz` to be installed.

Parameters **filename** : str or None, optional

The name (without an extension) of the file to write to disk. If *filename* is None, no file will be written, and we communicate with dot using only pipes.

format : {'png', 'pdf', 'dot', 'svg', 'jpeg', 'jpg'}, optional

Format in which to write output file. Default is 'png'.

optimize_graph : bool, optional

If True, the graph is optimized before rendering. Otherwise, the graph is displayed as is. Default is False.

****kwargs**

Additional keyword arguments to forward to `to_graphviz`.

Returns **result** : IPython.diplay.Image, IPython.display.SVG, or None

See dask.dot.dot_graph for more information.

See also:

`dask.base.visualize`, `dask.dot.dot_graph`

Notes

For more information on optimization see here:

http://dask.pydata.org/en/latest/optimize.html

where(*cond*, *other=nan*)

Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters **cond** : boolean NDFrame, array or callable

If cond is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as cond.

other : scalar, NDFrame, or callable

If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

try_cast : boolean, default False

try to cast the result back to the input type (if possible),

raise_on_error : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Returns **wh** : same type as caller

Notes

Dask doesn't supports following argument(s).

- inplace
- axis
- level
- try_cast
- raise_on_error

Series Methods

class dask.dataframe.**Series**
Out-of-core Series object

Mimics pandas.Series.

Parameters **dsk: dict**

The dask graph to compute this Series

_name: str

The key prefix that specifies which keys in the dask comprise this particular Series

name: scalar or None

Series name. This metadata aids usability

divisions: tuple of index values

Values along which we partition our blocks on the index

See also:

dask.dataframe.DataFrame

add(*other*, *level=None*, *fill_value=None*, *axis=0*)
Addition of series and other, element-wise (binary operator *add*).

Equivalent to series + other, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.radd

append(*other*)
Concatenate two or more Series.

Parameters **to_append** : Series or list/tuple of Series

verify_integrity : boolean, default False

If True, raise Exception on creating index with duplicates

Returns **appended** : Series

Notes

Dask doesn't supports following argument(s).

- to_append

•verify_integrity

Examples

```
>>> s1 = pd.Series([1, 2, 3])
>>> s2 = pd.Series([4, 5, 6])
>>> s3 = pd.Series([4, 5, 6], index=[3,4,5])
>>> s1.append(s2)
0    1
1    2
2    3
0    4
1    5
2    6
dtype: int64
```

```
>>> s1.append(s3)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With *verify_integrity* set to True:

```
>>> s1.append(s2, verify_integrity=True)
ValueError: Indexes have overlapping values: [0, 1, 2]
```

apply (*func*, *convert_dtype=True*, *name='__no_default__'*, *args=()*, ***kwds*)
Parallel version of pandas.Series.apply

This mimics the pandas version except for the following:

1.The user should provide output name.

Parameters **func: function**

Function to apply

convert_dtype: boolean, default True

Try to find better dtype for elementwise function results. If False, leave as dtype=object

name: list, scalar or None, optional

If list is given, the result is a DataFrame which columns is specified list. Otherwise, the result is a Series which name is given scalar or None (no name). If name keyword is not given, dask tries to infer the result type using its beginning of data. This inference may take some time and lead to unexpected result.

args: tuple

Positional arguments to pass to function in addition to the array/series

Additional keyword arguments will be passed as keywords to the function

Returns **applied** : Series or DataFrame depending on name keyword

astype(*dtype*)
Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters **dtype** : numpy.dtype or Python type

raise_on_error : raise on invalid input

kwargs : keyword arguments to pass on to the constructor

Returns **casted** : type of caller

Notes

Dask doesn't supports following argument(s).

•copy

•raise_on_error

between(*left*, *right*, *inclusive=True*)
Return boolean Series equivalent to left <= series <= right. NA values will be treated as False

Parameters **left** : scalar

Left boundary

right : scalar

Right boundary

Returns **is_between** : Series

cache(*cache=<type 'dict'>*)
Evaluate Dataframe and store in local cache

Uses chest by default to store data on disk

clip(*lower=None*, *upper=None*)
Trim values at input threshold(s).

Parameters **lower** : float or array_like, default None

upper : float or array_like, default None

axis : int or string axis name, optional

Align object with lower and upper along the given axis.

Returns **clipped** : Series

Notes

Dask doesn't supports following argument(s).

•axis

Examples

```
>>> df
  0         1
0  0.335232 -1.256177
1 -1.367855  0.746646
2  0.027753 -1.176076
3  0.230930 -0.679613
4  1.261967  0.570967
>>> df.clip(-1.0, 0.5)
          0         1
0  0.335232 -1.000000
1 -1.000000  0.500000
2  0.027753 -1.000000
3  0.230930 -0.679613
4  0.500000  0.500000
>>> t
0   -0.3
1   -0.2
2   -0.1
3    0.0
4    0.1
dtype: float64
>>> df.clip(t, t + 1, axis=0)
          0         1
0  0.335232 -0.300000
1 -0.200000  0.746646
2  0.027753 -0.100000
3  0.230930  0.000000
4  1.100000  0.570967
```

column_info
Return Series.name

compute(***kwargs*)
Compute several dask collections at once.

Parameters **get** : callable, optional

A scheduler `get` function to use. If not provided, the default is to check the global settings first, and then fall back to the collection defaults.

optimize_graph : bool, optional

If True [default], the graph is optimized before computation. Otherwise the graph is run as is. This can be useful for debugging.

kwargs

Extra keywords to forward to the scheduler `get` function.

corr(*other*, *method='pearson'*, *min_periods=None*)
Compute correlation with *other* Series, excluding missing values

Parameters **other** : Series

method : {'pearson', 'kendall', 'spearman'}

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods : int, optional

Minimum number of observations needed to have a valid result

Returns **correlation** : float

count()

Return number of non-NA/null observations in the Series

Parameters **level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

Returns **nobs** : int or Series (if level specified)

Notes

Dask doesn't supports following argument(s).

- level

cov(*other*, *min_periods=None*)

Compute covariance with Series, excluding missing values

Parameters **other** : Series

min_periods : int, optional

Minimum number of observations needed to have a valid result

Returns **covariance** : float

Normalized by N-1 (unbiased estimator).

cummax(*axis=None*, *skipna=True*)

Return cumulative cummax over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummax** : scalar

Notes

Dask doesn't supports following argument(s).

- dtype
- out

cummin(*axis=None*, *skipna=True*)

Return cumulative cummin over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummin** : scalar

Notes

Dask doesn't supports following argument(s).

- dtype
- out

cumprod(*axis=None*, *skipna=True*)

Return cumulative cumprod over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumprod** : scalar

Notes

Dask doesn't supports following argument(s).

- dtype
- out

cumsum(*axis=None*, *skipna=True*)

Return cumulative cumsum over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumsum** : scalar

Notes

Dask doesn't supports following argument(s).

- dtype
- out

describe()

Generate various summary statistics, excluding NaN values.

Parameters **percentiles** : array-like, optional

The percentiles to include in the output. Should all be in the interval [0, 1]. By default *percentiles* is [.25, .5, .75], returning the 25th, 50th, and 75th percentiles.

include, exclude : list-like, 'all', or None (default)

Specify the form of the returned result. Either:

- None to both (default). The result will include only numeric-typed columns or, if none are, only categorical columns.

- A list of dtypes or strings to be included/excluded. To select all numeric types use numpy numpy.number. To select categorical objects use type object. See also the select_dtypes documentation. eg. df.describe(include=['O'])
- If include is the string 'all', the output column-set will match the input one.

Returns summary: NDFrame of summary statistics

See also:

`DataFrame.select_dtypes`

Notes

Dask doesn't supports following argument(s).

•percentiles

•include

•exclude

div(*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.rtruediv

drop_duplicates(***kwargs*)
Return DataFrame with duplicate rows removed, optionally only considering certain columns

Parameters **subset** : column label or sequence of labels, optional

Only consider certain columns for identifying duplicates, by default use all of the columns

keep : {'first', 'last', False}, default 'first'

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

take_last : deprecated

inplace : boolean, default False

Whether to drop duplicates in place or to return a copy

Returns **deduplicated** : DataFrame

dropna()
Return Series without null values

Returns **valid** : Series

inplace : boolean, default False

Do operation in place.

Notes

Dask doesn't supports following argument(s).

•axis

•inplace

dtype
Return data type

fillna(*value*)
Fill NA/NaN values using the specified method

Parameters **value** : scalar, dict, Series, or DataFrame

Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0, 'index'}

inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit : int, default None

If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

downcast : dict, default is None

a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

Returns **filled** : Series

See also:

`reindex`, `asfreq`

Notes

Dask doesn't supports following argument(s).

- method
- axis
- inplace
- limit
- downcast

floordiv(*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division of series and other, element-wise (binary operator *floordiv*).

Equivalent to `series // other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.rfloordiv

get_division(*n*)

Get nth division of the data

groupby(*index*, ***kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

Parameters **by** : mapping function / list of functions, dict, Series, or tuple /

list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups

axis : int, default 0

level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort : boolean, default True

Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys : boolean, default True

When calling apply, add group keys to index to identify pieces

squeeze : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

Returns GroupBy object

Notes

Dask doesn't supports following argument(s).

- by
- axis
- level
- as_index
- sort
- group_keys
- squeeze

Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

head(*n=5*, *compute=True*)
First n rows of the dataset

Caveat, this only checks the first n rows of the first partition.

index
Return dask Index instance

isin(*other*)
Return a boolean `Series` showing whether each element in the `Series` is exactly contained in the passed sequence of `values`.

Parameters **values** : set or list-like

The sequence of values to test. Passing in a single string will raise a `TypeError`. Instead, turn a single string into a `list` of one element.

New in version 0.18.1.

Support for values as a set

Returns **isin** : Series (bool dtype)

Raises **TypeError**

- If `values` is a string

See also:

`pandas.DataFrame.isin`

Notes

Dask doesn't supports following argument(s).

•values

Examples

```
>>> s = pd.Series(list('abc'))
>>> s.isin(['a', 'c', 'e'])
0     True
1    False
2     True
dtype: bool
```

Passing a single string as `s.isin('a')` will raise an error. Use a list of one element instead:

```
>>> s.isin(['a'])
0     True
1    False
2    False
dtype: bool
```

isnull()
Return a boolean same-sized object indicating if the values are null.

See also:

notnull boolean inverse of isnull

iteritems()
Lazily iterate over (index, value) tuples

known_divisions
Whether divisions are already known

loc
Purely label-location based indexer for selection by label.

```
>>> df.loc["b"]
>>> df.loc["b":"d"]
```

map(*arg*, *na_action=None*)
Map values of Series using input correspondence (which can be a dict, Series, or function)

Parameters **arg** : function, dict, or Series

na_action : {None, 'ignore'}

If 'ignore', propagate NA values

Returns **y** : Series

same index as caller

Examples

```
>>> x
one   1
two   2
three 3
```

```
>>> y
1  foo
2  bar
3  baz
```

```
>>> x.map(y)
one   foo
two   bar
three baz
```

map_partitions (*func*, *columns='__no_default__'*, **args*, ***kwargs*)
Apply Python function on each DataFrame block

When using `map_partitions` you should provide either the column names (if the result is a DataFrame) or the name of the Series (if the result is a Series). The output type will be determined by the type of `columns`.

Parameters **func** : function

Function applied to each blocks

columns : tuple or scalar

Column names or name of the output. Defaults to names of data itself. When tuple is passed, DataFrame is returned. When scalar is passed, Series is returned.

Examples

When str is passed as columns, the result will be Series.

```
>>> df.map_partitions(lambda df: df.x + 1, columns='x')
```

When tuple is passed as columns, the result will be Series.

```
>>> df.map_partitions(lambda df: df.head(), columns=df.columns)
```

mask (*cond*, *other=nan*)
Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

Parameters **cond** : boolean NDFrame, array or callable

If cond is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as cond.

other : scalar, NDFrame, or callable

If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

try_cast : boolean, default False

try to cast the result back to the input type (if possible),

raise_on_error : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Returns **wh** : same type as caller

Notes

Dask doesn't supports following argument(s).

- inplace
- axis
- level
- try_cast
- raise_on_error

max(*axis=None*, *skipna=True*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **max** : scalar or Series (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

mean (*axis=None*, *skipna=True*)

Return the mean of the values for the requested axis

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **mean** : scalar or Series (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

min (*axis=None*, *skipna=True*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **min** : scalar or Series (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

mod(*other*, *level=None*, *fill_value=None*, *axis=0*)
Modulo of series and other, element-wise (binary operator *mod*).

Equivalent to `series % other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.rmod

mul(*other*, *level=None*, *fill_value=None*, *axis=0*)
Multiplication of series and other, element-wise (binary operator *mul*).

Equivalent to `series * other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.rmul

ndim
Return dimensionality

nlargest(*n=5*)
Return the largest *n* elements.

Parameters **n** : int

Return this many descending sorted values

keep : {'first', 'last', False}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

take_last : deprecated

Returns **top_n** : Series

The n largest values in the Series, in sorted order

See also:

`Series.nsmallest`

Notes

Faster than `.sort_values(ascending=False).head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(1e6))
>>> s.nlargest(10)  # only sorts up to the N requested
```

notnull()

Return a boolean same-sized object indicating if the values are not null.

See also:

isnull boolean inverse of notnull

npartitions

Return number of partitions

nunique()

Return number of unique elements in the object.

Excludes NA values by default.

Parameters **dropna** : boolean, default True

Don't include NaN in the count.

Returns **nunique** : int

Notes

Dask doesn't supports following argument(s).

- dropna

pow(*other*, *level=None*, *fill_value=None*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *pow*).

Equivalent to `series ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.rpow

quantile(*q=0.5*)
Approximate quantiles of Series

q [list/array of floats, default 0.5 (50%)] Iterable of numbers ranging from 0 to 1 for the desired quantiles

radd(*other*, *level=None*, *fill_value=None*, *axis=0*)
Addition of series and other, element-wise (binary operator *radd*).

Equivalent to `other + series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.add

random_split(*p*, *random_state=None*)
Pseudorandomly split dataframe into different pieces row-wise

Parameters **frac** : float, optional

Fraction of axis items to return.

random_state: int or np.random.RandomState

If int create a new RandomState with this as the seed

Otherwise draw from the passed RandomState

Examples

50/50 split

```
>>> a, b = df.random_split([0.5, 0.5])
```

80/10/10 split, consistent random_state

```
>>> a, b, c = df.random_split([0.8, 0.1, 0.1], random_state=123)
```

rdiv(*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.truediv

repartition(*divisions=None*, *npartitions=None*, *force=False*)
Repartition dataframe along new divisions

Parameters **divisions** : list

List of partitions to be used

npartitions : int

Number of partitions of output, must be less than npartitions of input

force : bool, default False

Allows the expansion of the existing divisions. If False then the new divisions lower and upper bounds must be the same as the old divisions.

Examples

```
>>> df = df.repartition(npartitions=10)
>>> df = df.repartition(divisions=[0, 5, 10, 20])
```

resample(*rule*, *how=None*, *closed=None*, *label=None*)
Convenience method for frequency conversion and resampling of regular time-series data.

Parameters **rule** : string

the offset string or object representing target conversion

axis : int, optional, default 0

closed : {'right', 'left'}

Which side of bin interval is closed

label : {'right', 'left'}

Which bin edge label to label bucket with

convention : {'start', 'end', 's', 'e'}

loffset : timedelta

Adjust the resampled time labels

base : int, default 0

For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

Notes

Dask doesn't supports following argument(s).

- axis
- fill_method
- convention
- kind
- loffset
- limit
- base

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label''2000-01-01 00:03:00'' does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
```

```
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00     0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00     1
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00     2
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

rfloordiv (*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:

Series.floordiv

rmod(*other*, *level=None*, *fill_value=None*, *axis=0*)
Modulo of series and other, element-wise (binary operator *rmod*).

Equivalent to `other % series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters other: Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:

Series.mod

rmul(*other*, *level=None*, *fill_value=None*, *axis=0*)
Multiplication of series and other, element-wise (binary operator *rmul*).

Equivalent to `other * series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters other: Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : Series

See also:

Series.mul

rolling(*window*, *min_periods=None*, *freq=None*, *center=False*, *win_type=None*, *axis=0*)
Provides rolling transformations.

Parameters window : int

Size of the moving window. This is the number of observations used for calculating the statistic. The window size must not be so large as to span more than one adjacent partition.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

center : boolean, default False

Set the labels at the center of the window.

win_type : string, default None

Provide a window type. The recognized window types are identical to pandas.

axis : int, default 0

Returns a Rolling object on which to call a method to compute a statistic

Notes

The *freq* argument is not supported.

rpow(*other*, *level=None*, *fill_value=None*, *axis=0*)
Exponential power of series and other, element-wise (binary operator *rpow*).

Equivalent to `other ** series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.pow

rsub(*other*, *level=None*, *fill_value=None*, *axis=0*)
Subtraction of series and other, element-wise (binary operator *rsub*).

Equivalent to `other - series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.sub

rtruediv (*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.truediv

sample (*frac*, *replace=False*, *random_state=None*)
Random sample of items

Parameters **frac** : float, optional

Fraction of axis items to return.

replace: boolean, optional

Sample with or without replacement. Default = False.

random_state: int or ‘‘np.random.RandomState‘‘

If int we create a new RandomState with this as the seed Otherwise we draw from the passed RandomState

std (*axis=None*, *ddof=1*, *skipna=True*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

ddof : int, default 1

degrees of freedom

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **std** : scalar or Series (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

sub(*other*, *level=None*, *fill_value=None*, *axis=0*)
Subtraction of series and other, element-wise (binary operator *sub*).

Equivalent to `series - other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.rsub

sum(*axis=None*, *skipna=True*)
Return the sum of the values for the requested axis

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **sum** : scalar or Series (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

tail(*n=5*, *compute=True*)
Last n rows of the dataset

Caveat, the only checks the last n rows of the last partition.

to_bag(*index=False*)
Convert to a dask Bag.

Parameters **index** : bool, optional

If True, the elements are tuples of `(index, value)`, otherwise they're just the `value`. Default is False.

to_csv(*filename*, *get=<function get_sync>*, ***kwargs*)
Write DataFrame to a comma-separated values (csv) file

Parameters **path_or_buf** : string or file handle, default None

File path or object, if None is provided the result is returned as a string.

sep : character, default ','

Field delimiter for the output file.

na_rep : string, default ''

Missing data representation

float_format : string, default None

Format string for floating point numbers

columns : sequence, optional

Columns to write

header : boolean or list of string, default True

Write out column names. If a list of string is given it is assumed to be aliases for the column names

index : boolean, default True

Write row names (index)

index_label : string or sequence, or False, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

nanRep : None

deprecated, use na_rep

mode : str

Python write mode, default 'w'

encoding : string, optional

A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

compression : string, optional

a string representing the compression to use in the output file, allowed values are 'gzip', 'bz2', 'xz', only used when the first argument is a filename

line_terminator : string, default 'n'

The newline character or character sequence to use in the output file

quoting : optional constant from csv module

defaults to csv.QUOTE_MINIMAL

quotechar : string (length 1), default ‘”’

character used to quote fields

doublequote : boolean, default True

Control quoting of *quotechar* inside a field

escapechar : string (length 1), default None

character used to escape *sep* and *quotechar* when appropriate

chunksize : int or None

rows to write at a time

tupleize_cols : boolean, default False

write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

date_format : string, default None

Format string for datetime objects

decimal: string, default ‘.’

Character recognized as decimal separator. E.g. use ‘,’ for European data

New in version 0.16.0.

Notes

Dask doesn’t supports following argument(s).

- path_or_buf
- sep
- na_rep
- float_format
- columns
- header
- index
- index_label
- mode
- encoding
- compression
- quoting
- quotechar
- line_terminator
- chunksize
- tupleize_cols

- date_format
- doublequote
- escapechar
- decimal

to_delayed()
Convert dataframe into dask Values

Returns a list of values, one value per partition.

to_frame(*name=None*)
Convert Series to DataFrame

Parameters **name** : object, default None

The passed name should substitute for the series name (if it has one).

Returns **data_frame** : DataFrame

to_hdf(*path_or_buf*, *key*, *mode='a'*, *append=False*, *complevel=0*, *complib=None*, *fletcher32=False*, *get=<function get_sync>*, ***kwargs*)
Activate the HDFStore.

Parameters **path_or_buf** : the path (string) or HDFStore object

key : string

indentifier for the group in the store

mode : optional, {'a', 'w', 'r', 'r+'}, default 'a'

'r' Read-only; no data can be modified.

'w' Write; a new file is created (an existing file with the same name would be deleted).

'a' Append; an existing file is opened for reading and writing, and if the file does not exist it is created.

'r+' It is similar to 'a', but the file must already exist.

format : 'fixed(f)|table(t)', default is 'fixed'

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable

table(t) [Table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data

append : boolean, default False

For Table formats, append the input data to the existing

complevel : int, 1-9, default 0

If a complib is specified compression will be applied where possible

complib : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dropna : boolean, default False.

If true, ALL nan rows will not be written to store.

truediv(*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters **other: Series or scalar value**

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

Series.rtruediv

unique()
Return Series of unique values in the object. Includes NA values.

Returns **uniques** : Series

value_counts()
Returns object containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

Parameters **normalize** : boolean, default False

If True then the object returned will contain the relative frequencies of the unique values.

sort : boolean, default True

Sort by values

ascending : boolean, default False

Sort in ascending order

bins : integer, optional

Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data

dropna : boolean, default True

Don't include counts of NaN.

Returns **counts** : Series

Notes

Dask doesn't supports following argument(s).

- normalize
- sort
- ascending

- bins
- dropna

var (*axis=None*, *ddof=1*, *skipna=True*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

ddof : int, default 1

degrees of freedom

numeric_only : boolean, default None

Include only float, int, boolean data. If None, will attempt to use everything, then use only numeric data

Returns **var** : scalar or Series (if level specified)

Notes

Dask doesn't supports following argument(s).

- level
- numeric_only

visualize (*filename='mydask'*, *format=None*, *optimize_graph=False*, ***kwargs*)
Render the computation of this object's task graph using graphviz.

Requires `graphviz` to be installed.

Parameters **filename** : str or None, optional

The name (without an extension) of the file to write to disk. If *filename* is None, no file will be written, and we communicate with dot using only pipes.

format : {'png', 'pdf', 'dot', 'svg', 'jpeg', 'jpg'}, optional

Format in which to write output file. Default is 'png'.

optimize_graph : bool, optional

If True, the graph is optimized before rendering. Otherwise, the graph is displayed as is. Default is False.

****kwargs**

Additional keyword arguments to forward to `to_graphviz`.

Returns **result** : IPython.diplay.Image, IPython.display.SVG, or None

See dask.dot.dot_graph for more information.

See also:

`dask.base.visualize`, `dask.dot.dot_graph`

Notes

For more information on optimization see here:

http://dask.pydata.org/en/latest/optimize.html

where(*cond*, *other=nan*)

Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters **cond** : boolean NDFrame, array or callable

If cond is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as cond.

other : scalar, NDFrame, or callable

If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1.

A callable can be used as other.

inplace : boolean, default False

Whether to perform the operation in place on the data

axis : alignment axis if needed, default None

level : alignment level if needed, default None

try_cast : boolean, default False

try to cast the result back to the input type (if possible),

raise_on_error : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Returns **wh** : same type as caller

Notes

Dask doesn't supports following argument(s).

- inplace
- axis
- level
- try_cast
- raise_on_error

DataFrameGroupBy

class dask.dataframe.groupby.**DataFrameGroupBy**(*df*, *index=None*, *slice=None*, ***kwargs*)

apply(*func*, *columns='__no_default__'*)
Parallel version of pandas GroupBy.apply

This mimics the pandas version except for the following:

1. The user should provide output columns.
2. If the grouper does not align with the index then this causes a full shuffle. The order of rows within each group may not be preserved.

Parameters **func: function**

Function to apply

columns: list, scalar or None

If list is given, the result is a DataFrame which columns is specified list. Otherwise, the result is a Series which name is given scalar or None (no name). If name keyword is not given, dask tries to infer the result type using its beginning of data. This inference may take some time and lead to unexpected result

Returns **applied** : Series or DataFrame depending on columns keyword

count()
Compute count of group, excluding missing values

See also:

pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

get_group(*key*)
Constructs NDFrame from group with provided name

Parameters **name** : object

the name of the group to get as a DataFrame

obj : NDFrame, default None

the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

Returns **group** : type of obj

Notes

Dask doesn't supports following argument(s).

- name
- obj

max()
Compute max of group values

See also:

pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

mean()
Compute mean of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

min()
Compute min of group values

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

std(*ddof=1*)
Compute standard deviation of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

Parameters **ddof** : integer, default 1

degrees of freedom

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

sum()
Compute sum of group values

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

var(*ddof=1*)
Compute variance of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

Parameters **ddof** : integer, default 1

degrees of freedom

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

SeriesGroupBy

class `dask.dataframe.groupby.`**SeriesGroupBy**(*df*, *index*, *slice=None*, ***kwargs*)

apply(*func*, *columns='__no_default__'*)
Parallel version of pandas GroupBy.apply

This mimics the pandas version except for the following:

1. The user should provide output columns.
2. If the grouper does not align with the index then this causes a full shuffle. The order of rows within each group may not be preserved.

Parameters **func: function**

Function to apply

columns: list, scalar or None

If list is given, the result is a DataFrame which columns is specified list. Otherwise, the result is a Series which name is given scalar or None (no name). If name keyword is not given, dask tries to infer the result type using its beginning of data. This inference may take some time and lead to unexpected result

Returns applied : Series or DataFrame depending on columns keyword

count()
Compute count of group, excluding missing values

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

get_group(*key*)
Constructs NDFrame from group with provided name

Parameters name : object

the name of the group to get as a DataFrame

obj : NDFrame, default None

the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

Returns group : type of obj

Notes

Dask doesn't supports following argument(s).

- name
- obj

max()
Compute max of group values

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

mean()
Compute mean of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

min()
Compute min of group values

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

std(*ddof=1*)
Compute standard deviation of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

Parameters **ddof** : integer, default 1

degrees of freedom

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

sum()
Compute sum of group values

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

var(*ddof=1*)
Compute variance of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

Parameters **ddof** : integer, default 1

degrees of freedom

See also:

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

Other functions

`dask.dataframe.`**compute**(**args*, ***kwargs*)
Compute several dask collections at once.

Parameters **args** : object

Any number of objects. If the object is a dask collection, it's computed and the result is returned. Otherwise it's passed through unchanged.

get : callable, optional

A scheduler `get` function to use. If not provided, the default is to check the global settings first, and then fall back to defaults for the collections.

optimize_graph : bool, optional

If True [default], the optimizations for each collection are applied before computation. Otherwise the graph is run as is. This can be useful for debugging.

kwargs

Extra keywords to forward to the scheduler `get` function.

Examples

```
>>> import dask.array as da
>>> a = da.arange(10, chunks=2).sum()
>>> b = da.arange(10, chunks=2).mean()
```

```
>>> compute(a, b)
(45, 4.5)
```

dask.dataframe.**map_partitions**(*func*, *metadata*, **args*, ***kwargs*)
Apply Python function on each DataFrame block

Parameters **metadata: _Frame, columns, name**

Metadata for output

targets : list

List of target DataFrame / Series.

dask.dataframe.multi.**concat**(*dfs*, *axis=0*, *join='outer'*, *interleave_partitions=False*)
Concatenate DataFrames along rows.

- When axis=0 (default), concatenate DataFrames row-wise:
  - If all divisions are known and ordered, concatenate DataFrames keeping divisions. When divisions are not ordered, specifying interleave_partition=True allows concatenate divisions each by each.
  - If any of division is unknown, concatenate DataFrames resetting its division to unknown (None)
- When axis=1, concatenate DataFrames column-wise:
  - Allowed if all divisions are known.
  - If any of division is unknown, it raises ValueError.

Parameters **dfs** : list

List of dask.DataFrames to be concatenated

axis : {0, 1, 'index', 'columns'}, default 0

The axis to concatenate along

join : {'inner', 'outer'}, default 'outer'

How to handle indexes on other axis

interleave_partitions : bool, default False

Whether to concatenate DataFrames ignoring its order. If True, every divisions are concatenated each by each.

Examples

If all divisions are known and ordered, divisions are kept.

```
>>> a
dd.DataFrame<x, divisions=(1, 3, 5)>
>>> b
dd.DataFrame<y, divisions=(6, 8, 10)>
>>> dd.concat([a, b])
dd.DataFrame<concat-..., divisions=(1, 3, 6, 8, 10)>
```

Unable to concatenate if divisions are not ordered.

```
>>> a
dd.DataFrame<x, divisions=(1, 3, 5)>
>>> b
dd.DataFrame<y, divisions=(2, 3, 6)>
```

```
>>> dd.concat([a, b])
ValueError: All inputs have known divisions which cannot be concatenated
in order. Specify interleave_partitions=True to ignore order
```

Specify interleave_partitions=True to ignore the division order.

```
>>> dd.concat([a, b], interleave_partitions=True)
dd.DataFrame<concat-..., divisions=(1, 2, 3, 5, 6)>
```

If any of division is unknown, the result division will be unknown

```
>>> a
dd.DataFrame<x, divisions=(None, None)>
>>> b
dd.DataFrame<y, divisions=(1, 4, 10)>
>>> dd.concat([a, b])
dd.DataFrame<concat-..., divisions=(None, None, None, None)>
```

dask.dataframe.multi.**merge**(*left*, *right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *suffixes=('_x', '_y')*, *npartitions=None*)

dask.dataframe.**read_csv**(*urlpath*, *blocksize=33554432*, *chunkbytes=None*, *collection=True*, *lineterminator=None*, *compression=None*, *sample=256000*, *enforce=False*, *storage_options=None*, ***kwargs*)

Read CSV files into a Dask.DataFrame

This parallelizes the `pandas.read_csv` file in the following ways:

1.It supports loading many files at once using globstrings as follows:

```
>>> df = dd.read_csv('myfiles.*.csv')
```

2.In some cases it can break up large files as follows:

```
>>> df = dd.read_csv('largefile.csv', blocksize=25e6)  # 25MB chunks
```

3.You can read CSV files from external resources (e.g. S3, HDFS) providing a URL:

```
>>> df = dd.read_csv('s3://bucket/myfiles.*.csv')
>>> df = dd.read_csv('hdfs:///myfiles.*.csv')
>>> df = dd.read_csv('hdfs://namenode.example.com/myfiles.*.csv')
```

Internally dd.read_csv uses pandas.read_csv and so supports many of the same keyword arguments with the same performance guarantees.

See the docstring for `pandas.read_csv` for more information on available keyword arguments.

Note that this function may fail if a CSV file includes quoted strings that contain the line terminator.

Parameters **urlpath: string**

Absolute or relative filepath, URL (may include protocols like `s3://`), or globstring for CSV files.

blocksize: int or None

Number of bytes by which to cut up larger files. Default value is computed based on available physical memory and the number of cores. If `None`, use a single block for each file.

collection: boolean

Return a dask.dataframe if True or list of dask.delayed objects if False

sample: int

Number of bytes to use when determining dtypes

storage_options: dict

Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

****kwargs: dict**

Options to pass down to `pandas.read_csv`

`dask.dataframe.`**`from_array`**(*x*, *chunksize=50000*, *columns=None*)

Read dask Dataframe from any slicable array

Uses getitem syntax to pull slices out of the array. The array need not be a NumPy array but must support slicing syntax

x[50000:100000]

and have 2 dimensions:

x.ndim == 2

or have a record dtype:

x.dtype == [('name', 'O'), ('balance', 'i8')]

`dask.dataframe.`**`from_pandas`**(*data*, *npartitions=None*, *chunksize=None*, *sort=True*, *name=None*)

Construct a dask object from a pandas object.

If given a `pandas.Series` a `dask.Series` will be returned. If given a `pandas.DataFrame` a `dask.DataFrame` will be returned. All other pandas objects will raise a `TypeError`.

Parameters **df** : pandas.DataFrame or pandas.Series

The DataFrame/Series with which to construct a dask DataFrame/Series

npartitions : int, optional

The number of partitions of the index to create.

chunksize : int, optional

The size of the partitions of the index.

Returns dask.DataFrame or dask.Series

A dask DataFrame/Series partitioned along the index

Raises **TypeError**

If something other than a `pandas.DataFrame` or `pandas.Series` is passed in.

See also:

`from_array` Construct a dask.DataFrame from an array that has record dtype

`from_bcolz` Construct a dask.DataFrame from a bcolz ctable

`read_csv` Construct a dask.DataFrame from a CSV file

Examples

```
>>> df = pd.DataFrame(dict(a=list('aabbcc'), b=list(range(6))),
...                   index=pd.date_range(start='20100101', periods=6))
>>> ddf = from_pandas(df, npartitions=3)
>>> ddf.divisions
(Timestamp('2010-01-01 00:00:00', offset='D'),
 Timestamp('2010-01-03 00:00:00', offset='D'),
 Timestamp('2010-01-05 00:00:00', offset='D'),
 Timestamp('2010-01-06 00:00:00', offset='D'))
>>> ddf = from_pandas(df.a, npartitions=3)  # Works with Series too!
>>> ddf.divisions
(Timestamp('2010-01-01 00:00:00', offset='D'),
 Timestamp('2010-01-03 00:00:00', offset='D'),
 Timestamp('2010-01-05 00:00:00', offset='D'),
 Timestamp('2010-01-06 00:00:00', offset='D'))
```

dask.dataframe.**from_bcolz**(*x*, *chunksize=None*, *categorize=True*, *index=None*, *lock=<thread.lock object>*, ***kwargs*)

Read dask Dataframe from bcolz.ctable

Parameters **x** : bcolz.ctable

Input data

chunksize : int, optional

The size of blocks to pull out from ctable. Ideally as large as can comfortably fit in memory

categorize : bool, defaults to True

Automatically categorize all string dtypes

index : string, optional

Column to make the index

lock: bool or Lock

Lock to use when reading or False for no lock (not-thread-safe)

See also:

from_array more generic function not optimized for bcolz

dask.dataframe.rolling.**rolling_apply**(*arg*, *window*, **args*, ***kwargs*)

Generic moving function application.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

func : function

Must produce a single value from an ndarray input

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Whether the label should correspond with center of window

args : tuple

Passed on to func

kwargs : dict

Passed on to func

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_chunk`**(*func*, *part1*, *part2*, *window*, **args*)

`dask.dataframe.rolling.`**`rolling_count`**(*arg*, *window*, **args*, ***kwargs*)

Rolling count of number of non-NaN observations inside provided window.

Parameters **arg** : DataFrame or numpy ndarray-like

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Whether the label should correspond with center of window

how : string, default 'mean'

Method for down- or re-sampling

Returns **rolling_count** : type of caller

Notes

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_kurt`**(*arg*, *window*, **args*, ***kwargs*)

Unbiased moving kurtosis.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default 'None'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_max`**(*arg*, *window*, **args*, ***kwargs*)

Moving maximum.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default ''max'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_mean`**(*arg*, *window*, **args*, ***kwargs*)
Moving mean.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default 'None'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_median`**(*arg*, *window*, **args*, ***kwargs*)
Moving median.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default ''median'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_min`**(*arg*, *window*, **args*, ***kwargs*)

Moving minimum.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default ''min'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_quantile`**(*arg*, *window*, **args*, ***kwargs*)

Moving quantile.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

quantile : float

0 <= quantile <= 1

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Whether the label should correspond with center of window

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_skew`**(*arg*, *window*, **args*, ***kwargs*)

Unbiased moving skewness.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default 'None'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_std`**(*arg*, *window*, **args*, ***kwargs*)

Moving standard deviation.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default 'None'

Method for down- or re-sampling

ddof : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_sum`**(*arg*, *window*, **args*, ***kwargs*)

Moving sum.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default 'None'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_var`**(*arg*, *window*, **args*, ***kwargs*)

Moving variance.

Parameters **arg** : Series, DataFrame

window : int

Size of the moving window. This is the number of observations used for calculating the statistic.

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

how : string, default 'None'

Method for down- or re-sampling

ddof : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements.

Returns **y** : type of input argument

Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

`dask.dataframe.rolling.`**`rolling_window`**(*arg*, *window*, **args*, ***kwargs*)

Applies a moving window of type `window_type` and size `window` on the data.

Parameters **arg** : Series, DataFrame

window : int or ndarray

Weighting window specification. If the window is an integer, then it is treated as the window length and win_type is required

win_type : str, default None

Window type (see Notes)

min_periods : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Whether the label should correspond with center of window

mean : boolean, default True

If True computes weighted mean, else weighted sum

axis : {0, 1}, default 0

how : string, default 'mean'

Method for down- or re-sampling

Returns **y** : type of input argument

Notes

The recognized window types are:

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`

- `blackmanharris`
- `nuttall`
- `barthann`
- `kaiser` (needs beta)
- `gaussian` (needs std)
- `general_gaussian` (needs power, width)
- `slepian` (needs width).

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

4.6.4 Examples

Dataframes from CSV files

Suppose we have a collection of CSV files with data:

data1.csv:

```
time,temperature,humidity
0,22,58
1,21,57
2,25,57
3,26,55
4,22,53
5,23,59
```

data2.csv:

```
time,temperature,humidity
0,24,85
1,26,83
2,27,85
3,25,92
4,25,83
5,23,81
```

data3.csv:

```
time,temperature,humidity
0,18,51
1,15,57
2,18,55
3,19,51
4,19,52
5,19,57
```

and so on.

We can create Dask dataframes from CSV files using `dd.read_csv`.

```
>>> import dask.dataframe as dd
>>> df = dd.read_csv('data*.csv')
```

We can work with the Dask dataframe as usual, which is composed of Pandas dataframes. We can list the first few rows.

```
>>> df.head()
time  temperature  humidity
0     0            22        58
1     1            21        57
2     2            25        57
3     3            26        55
4     4            22        53
```

Or we can compute values over the entire dataframe.

```
>>> df.temperature.mean().compute()
22.055555555555557

>>> df.humidity.std().compute()
14.710829233324224
```

Other topics

4.6.5 Partitions

Internally a dask dataframe is split into many partitions, and each partition is one pandas dataframe. These dataframes are split vertically along the index. When our index is sorted and we know the values of the divisions of our partitions, then we can be clever and efficient.

For example, if we have a time-series index then our partitions might be divided by month. All of January will live in one partition while all of February will live in the next. In these cases operations like `loc`, `groupby`, and `join/merge` along the index can be *much* more efficient than would otherwise be possible in parallel. You can view the number of partitions and divisions of your dataframe with the following fields:

```
>>> df.npartitions
4
>>> df.divisions
['2015-01-01', '2015-02-01', '2015-03-01', '2015-04-01', '2015-04-31']
```

Divisions includes the minimum value of every partition's index and the maximum value of the last partition's index. In the example above if the user searches for a specific datetime range then we know which partitions we need to inspect and which we can drop:

```
>>> df.loc['2015-01-20': '2015-02-10']  # Must inspect first two partitions
```

Often we do not have such information about our partitions. When reading CSV files for example we do not know, without extra user input, how the data is divided. In this case `.divisions` will be all `None`:

```
>>> df.divisions
[None, None, None, None, None]
```

4.6.6 Shuffling for GroupBy and Join

Operations like `groupby`, `join`, and `set_index` have special performance considerations that are different from normal Pandas due to the parallel, larger-than-memory, and distributed nature of dask.dataframe.

Easy Case

To start off, common groupby operations like `df.groupby(columns).reduction()` for known reductions like `mean, sum, std, var, count, nunique` are all quite fast and efficient. This is the common case. So too with joins. Joining a Dask dataframe to a Pandas dataframe or joining two Dask dataframes along their index is already quite fast. No special considerations need to be made when operating in these common cases.

So if you're doing common groupby and join operations then you can stop reading this. Everything will scale nicely. Fortunately this is true most of the time.

```
>>> df.groupby(columns).known_reduction()     # Fast and common case
>>> dask_df.join(pandas_df, on=column)        # Fast and common case
```

Difficult Cases

In some cases, such as when applying an arbitrary function to groups, when joining along non-index columns, or when explicitly setting an unsorted column to be the index, we may need to trigger a full dataset shuffle

```
>>> df.groupby(columns).apply(arbitrary_user_function)  # Requires shuffle
>>> lhs.join(rhs, on=column)                            # Requires shuffle
>>> df.set_index(column)                                # Requires shuffle
```

A shuffle is necessary when we need to re-sort our data along a new index. For example if we have banking records that are organized by time and we now want to organize them by user ID then we'll need to move a lot of data around. In Pandas all of this data fit in memory, so this operation was easy. Now that we don't assume that all data fits in memory we must be a bit more careful.

Re-sorting the data can be avoided by restricting yourself to the easy cases mentioned above.

Shuffle Methods

There are currently two strategies to shuffle data depending on whether you are on a single machine or on a distributed cluster.

Shuffle on Disk

When operating on larger-than-memory data on a single machine we shuffle by dumping intermediate results to disk. This is done using the partd project for on-disk shuffles.

Shuffle over the Network

When operating on a distributed cluster the Dask workers may not have access to a shared hard drive. In this case we shuffle data by breaking input partitions into many pieces based on where they will end up and moving these pieces throughout the network. This prolific expansion of intermediate partitions can stress the task scheduler. To manage for many-partitioned datasets this we sometimes shuffle in stages, causing undue copies but reducing the `n**2` effect of shuffling to something closer to `n log(n)` with `log(n)` copies.

Selecting methods

Dask will use on-disk shuffling by default but will switch to task-based distributed shuffling if the default scheduler is set to use a `dask.distributed.Executor` such as would be the case if the user sets the Executor as default using one of the following two options:

```
e = Executor('scheduler:8786', set_as_default=True)

or

dask.set_options(get=e.get)
```

Alternatively, if you prefer to avoid defaults, you can specify a `method=` keyword argument to `groupby` or `set_index`

```
df.set_index(column, method='disk')
df.set_index(column, method='tasks')
```

4.7 Delayed

As discussed in the *custom graphs* section, sometimes a problem doesn't fit into one of the collections like `dask.bag` or `dask.array`. Instead of creating a dask directly using a dictionary, one can use the `dask.delayed` interface. This allows one to create graphs directly with a light annotation of normal python code.

4.7.1 Overview

Motivation and Example

Dask.delayed lets you parallelize custom code. It is useful whenever your problem doesn't quite fit a high-level parallel object like dask.array or dask.dataframe but could still benefit from parallelism. Dask.delayed works by delaying your function evaluations and putting them into a dask graph. Dask.delayed is useful when wrapping existing code or when handling non-standard problems.

Consider the following example:

```
def inc(x):
    return x + 1

def double(x):
    return x + 2

def add(x, y):
    return x + y

data = [1, 2, 3, 4, 5]

output = []
for x in data:
    a = inc(x)
    b = double(x)
    c = add(a, b)
    output.append(c)

total = sum(output)
```

As written this code runs sequentially in a single thread. However we see that a lot of this could be executed in parallel. We use the `delayed` function to parallelize this code by turning it into a dask graph. We slightly modify our code by wrapping functions in `delayed`. This delays the execution of the function and generates a dask graph instead.

```
from dask import delayed

 output = []
 for x in data:
     a = delayed(inc)(x)
     b = delayed(double)(x)
     c = delayed(add)(a, b)
     output.append(c)

 total = delayed(sum)(output)
```

We used the `delayed` function to wrap the function calls that we want to turn into tasks. None of the `inc`, `double`, `add` or `sum` calls have happened yet, instead the object `total` is a `Delayed` result that contains a task graph of the entire computation. Looking at the graph we see clear opportunities for parallel execution. The dask schedulers will exploit this parallelism, generally improving performance. (although not in this example, because these functions are already very small and fast.)

```
total.visualize()  # see image to the right
```

We can now compute this lazy result to execute the graph in parallel:

```
>>> total.compute()
45
```

Delayed Function

The `dask.delayed` interface consists of one function, `delayed`:

- `delayed` wraps functions

  Wraps functions. Can be used as a decorator, or around function calls directly (i.e. `delayed(foo)(a, b, c)`). Outputs from functions wrapped in `delayed` are proxy objects of type `Delayed` that contain a graph of all operations done to get to this result.

- `delayed` wraps objects

  Wraps objects. Used to create `Delayed` proxies directly.

`Delayed` objects can be thought of as representing a key in the dask. A `Delayed` supports *most* python operations, each of which creates another `Delayed` representing the result:

- Most operators (`*`, `-`, and so on)
- Item access and slicing (`a[0]`)
- Attribute access (`a.size`)
- Method calls (`a.index(0)`)

Operations that aren't supported include:

- Mutating operators (`a += 1`)
- Mutating magics such as `__setitem__`/`__setattr__` (`a[0] = 1`, `a.foo = 1`)
- Iteration. (`for i in a: ...`)
- Use as a predicate (`if a: ...`)

The last two points in particular mean that `Delayed` objects cannot be used for control flow, meaning that no `Delayed` can appear in a loop or if statement. In other words you can't iterate over a `Delayed` object, or use it as part of a condition in an if statement, but `Delayed` object can be used in a body of a loop or if statement (i.e. the example above is fine, but if `data` was a `Delayed` object it wouldn't be). Even with this limitation, many workflows can easily be parallelized.

4.7.2 API

delayed	Wraps a function or object to produce a `Delayed`.
compute(*args, **kwargs)	Evaluate more than one `Delayed` at once.

dask.delayed.**delayed**()

Wraps a function or object to produce a `Delayed`.

`Delayed` objects act as proxies for the object they wrap, but all operations on them are done lazily by building up a dask graph internally.

Parameters **obj** : object

The function or object to wrap

name : string or hashable, optional

The key to use in the underlying graph for the wrapped object. Defaults to hashing content.

pure : bool, optional

Indicates whether calling the resulting `Delayed` object is a pure operation. If True, arguments to the call are hashed to produce deterministic keys. Default is False.

Examples

Apply to functions to delay execution:

```
>>> def inc(x):
...     return x + 1
```

```
>>> inc(10)
11
```

```
>>> x = delayed(inc, pure=True)(10)
>>> type(x) == Delayed
True
>>> x.compute()
11
```

Can be used as a decorator:

```
>>> @delayed(pure=True)
... def add(a, b):
...     return a + b
>>> add(1, 2).compute()
3
```

`delayed` also accepts an optional keyword `pure`. If False (default), then subsequent calls will always produce a different `Delayed`. This is useful for non-pure functions (such as `time` or `random`).

```
>>> from random import random
>>> out1 = delayed(random, pure=False)()
>>> out2 = delayed(random, pure=False)()
>>> out1.key == out2.key
False
```

If you know a function is pure (output only depends on the input, with no global state), then you can set `pure=True`. This will attempt to apply a consistent name to the output, but will fallback on the same behavior of `pure=False` if this fails.

```
>>> @delayed(pure=True)
... def add(a, b):
...     return a + b
>>> out1 = add(1, 2)
>>> out2 = add(1, 2)
>>> out1.key == out2.key
True
```

The key name of the result of calling a delayed object is determined by hashing the arguments by default. To explicitly set the name, you can use the `dask_key_name` keyword when calling the function:

```
>>> add(1, 2)
Delayed('add-3dce7c56edd1ac2614add714086e950f')
>>> add(1, 2, dask_key_name='three')
Delayed('three')
```

Note that objects with the same key name are assumed to have the same result. If you set the names explicitly you should make sure your key names are different for different results.

```
>>> add(1, 2, dask_key_name='three')
>>> add(2, 1, dask_key_name='three')
>>> add(2, 2, dask_key_name='four')
```

`delayed` can also be applied to objects to make operations on them lazy:

```
>>> a = delayed([1, 2, 3])
>>> isinstance(a, Delayed)
True
>>> a.compute()
[1, 2, 3]
```

The key name of a delayed object is hashed by default. To explicitly set the name, you can use the `name` keyword:

```
>>> a = delayed([1, 2, 3], name='mylist')
>>> a
Delayed('mylist')
```

Delayed results act as a proxy to the underlying object. Many operators are supported:

```
>>> (a + [1, 2]).compute()
[1, 2, 3, 1, 2]
>>> a[1].compute()
2
```

Method and attribute access also works:

```
>>> a.count(2).compute()
1
```

Note that if a method doesn't exist, no error will be thrown until runtime:

```
>>> res = a.not_a_real_method()
>>> res.compute()
AttributeError("'list' object has no attribute 'not_a_real_method'")
```

Methods are assumed to be impure by default, meaning that subsequent calls may return different results. To assume purity, set *pure=True*. This allows sharing of any intermediate values.

```
>>> a.count(2, pure=True).key == a.count(2, pure=True).key
True
```

As with function calls, method calls also support the `dask_key_name` keyword:

```
>>> a.count(2, dask_key_name="count_2")
Delayed("count_2")
```

`dask.delayed.`**`compute`**(**args*, ***kwargs*)

Evaluate more than one `Delayed` at once.

Note that the only difference between this function and `dask.base.compute` is that this implicitly wraps python objects in `Delayed`, allowing for collections of dask objects to be computed.

Examples

```
>>> a = value(1)
>>> b = a + 2
>>> c = a + 3
>>> compute(b, c)  # Compute both simultaneously
(3, 4)
>>> compute(a, [b, c])  # Works for lists of Delayed
(1, [3, 4])
```

4.7.3 Examples

Build Custom Arrays

Here we have a serial blocked computation for computing the mean of all positive elements in a large, on disk array:

```
x = h5py.File('myfile.hdf5')['/x']          # Trillion element array on disk

sums = []
counts = []
for i in range(1000000):                    # One million times
    chunk = x[1000000*i:1000000*(i + 1)]    # Pull out chunk
    positive = chunk[chunk > 0]             # Filter out negative elements
    sums.append(positive.sum())             # Sum chunk
    counts.append(positive.size)            # Count chunk

result = sum(sums) / sum(counts)            # Aggregate results
```

Below is the same code, parallelized using `dask.delayed`:

```
x = delayed(h5py.File('myfile.hdf5')['/x'])   # Trillion element array on disk

sums = []
counts = []
for i in range(1000000):                    # One million times
```

```
    chunk = x[1000000*i:1000000*(i + 1)]     # Pull out chunk
    positive = chunk[chunk > 0]              # Filter out negative elements
    sums.append(positive.sum())              # Sum chunk
    counts.append(positive.size)             # Count chunk

result = delayed(sum)(sums) / delayed(sum)(counts)     # Aggregate results

result.compute()                             # Perform the computation
```

Only 3 lines had to change to make this computation parallel instead of serial.

- Wrap the original array in `delayed`. This makes all the slices on it return `Delayed` objects.
- Wrap both calls to `sum` with `delayed`.
- Call the `compute` method on the result.

While the for loop above still iterates fully, it's just building up a graph of the computation that needs to happen, without actually doing any computing.

Data Processing Pipelines

Example notebook.

Now, rebuilding the example from *custom graphs*:

```
from dask import delayed, value

@delayed
def load(filename):
    ...

@delayed
def clean(data):
    ...

@delayed
def analyze(sequence_of_data):
    ...

@delayed
def store(result):
    with open(..., 'w') as f:
        f.write(result)

files = ['myfile.a.data', 'myfile.b.data', 'myfile.c.data']
loaded = [load(i) for i in files]
cleaned = [clean(i) for i in loaded]
analyzed = analyze(cleaned)
stored = store(analyzed)

stored.compute()
```

This builds the same graph as seen before, but using normal Python syntax. In fact, the only difference between Python code that would do this in serial, and the parallel version with dask is the `delayed` decorators on the functions, and the call to `compute` at the end.

4.7.4 Working with Collections

Often we want to do a bit of custom work with `dask.delayed` (for example for complex data ingest), then leverage the algorithms in `dask.array` or `dask.dataframe`, and then switch back to custom work. To this end, all collections support `from_delayed` functions and `to_delayed` methods.

As an example, consider the case where we store tabular data in a custom format not known by `dask.dataframe`. This format is naturally broken apart into pieces and we have a function that reads one piece into a Pandas DataFrame. We use `dask.delayed` to lazily read these files into Pandas DataFrames, use `dd.from_delayed` to wrap these pieces up into a single `dask.dataframe`, use the complex algorithms within `dask.dataframe` (groupby, join, etc..) and then switch back to delayed to save our results back to the custom format.

```
import dask.dataframe as dd
from dask.delayed import delayed

from my_custom_library import load, save

filenames = ...
dfs = [delayed(load)(fn) for fn in filenames]

df = dd.from_delayed(dfs)
df = ... # do work with dask.dataframe

dfs = df.to_delayed()
writes = [delayed(save)(df, fn) for df, fn in zip(dfs, filenames)]

dd.compute(*writes)
```

Data science is often complex, `dask.delayed` provides a release valve for users to manage this complexity on their own, and solve the last mile problem for custom formats and complex situations.

Graphs

Dask graphs encode algorithms in a simple format involving Python dicts, tuples, and functions. This graph format can be used in isolation from the dask collections. Working directly with dask graphs is an excellent way to implement and test new algorithms in fields such as linear algebra, optimization, and machine learning. If you are a *developer*, you should start here.

- Overview
- Specification
- Custom Graphs
- Optimization

4.8 Overview

An explanation of dask task graphs.

4.8.1 Motivation

Normally, humans write programs and then compilers/interpreters interpret them (for example `python`, `javac`, `clang`). Sometimes humans disagree with how these compilers/interpreters choose to interpret and execute their programs. In these cases humans often bring the analysis, optimization, and execution of code into the code itself.

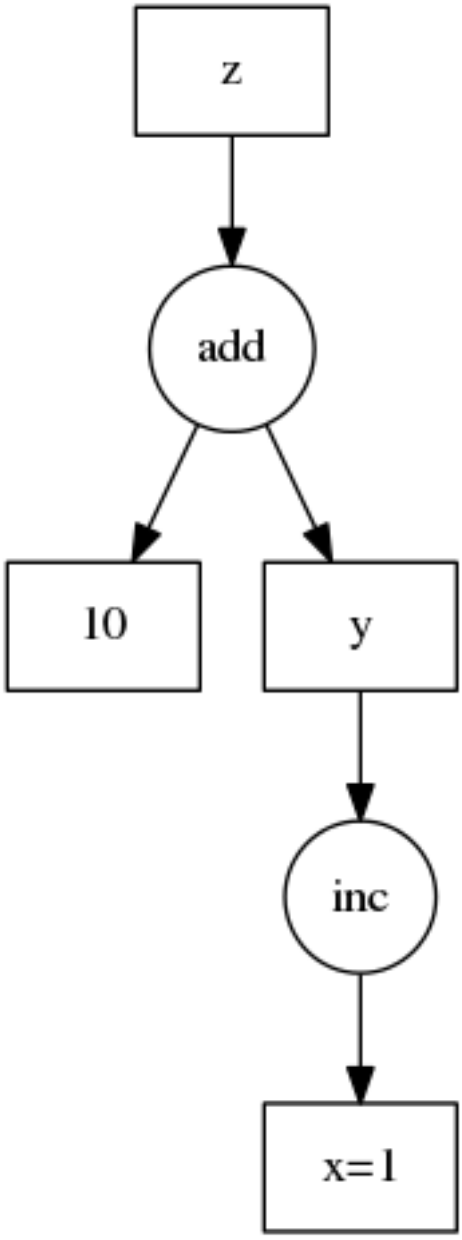
Commonly a desire for parallel execution causes this shift of responsibility from compiler to human developer. In these cases, we often represent the structure of our program explicitly as data within the program itself.

A common approach to parallel execution in user-space is *task scheduling*. In task scheduling we break our program into many medium-sized tasks or units of computation, often a function call on a non-trivial amount of data. We represent these tasks as nodes in a graph with edges between nodes if one task depends on data produced by another. We call upon a *task scheduler* to execute this graph in a way that respects these data dependencies and leverages parallelism where possible, multiple independent tasks can be run simultaneously.

Many solutions exist. This is a common approach in parallel execution frameworks. Often task scheduling logic hides within other larger frameworks (Luigi, Storm, Spark, IPython Parallel, and so on) and so is often reinvented.

Dask is a specification that encodes task schedules with minimal incidental complexity using terms common to all Python projects, namely dicts, tuples, and callables. Ideally this minimum solution is easy to adopt and understand by a broad community.

4.8.2 Example



Consider the following simple program:

```
def inc(i):
    return i + 1

def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

We encode this as a dictionary in the following way:

```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

While less pleasant than our original code, this representation can be analyzed and executed by other Python code, not just the CPython interpreter. We don't recommend that users write code in this way, but rather that it is an appropriate target for automated systems. Also, in non-toy examples, the execution times are likely much larger than for `inc` and `add`, warranting the extra complexity.

4.8.3 Schedulers

The `dask` library currently contains a few schedulers to execute these graphs. Each scheduler works differently, providing different performance guarantees and operating in different contexts. These implementations are not special and others can write different schedulers better suited to other applications or architectures easily. Systems that emit dask graphs (like `dask.array`, `dask.bag`, and so on) may leverage the appropriate scheduler for the application and hardware.

4.9 Specification

Dask is a specification to encode a graph – specifically, a directed acyclic graph of tasks with data dependencies – using ordinary Python data structures, namely dicts, tuples, functions, and arbitrary Python values.

4.9.1 Definitions

A **dask graph** is a dictionary mapping **keys** to **computations**:

```
{'x': 1,
 'y': 2,
 'z': (add, 'x', 'y'),
 'w': (sum, ['x', 'y', 'z']),
 'v': [(sum, ['w', 'z']), 2]}
```

A **key** is any hashable value that is not a **task**:

```
'x'
('x', 2, 3)
```

A **task** is a tuple with a callable first element. Tasks represent atomic units of work meant to be run by a single worker. Example:

```
(add, 'x', 'y')
```

We represent a task as a tuple such that the *first element is a callable function* (like `add`), and the succeeding elements are *arguments* for that function. An *argument* may be any valid **computation**.

A **computation** may be one of the following:

1. Any **key** present in the dask graph like `'x'`
2. Any other value like `1`, to be interpreted literally
3. A **task** like `(inc, 'x')` (see below)
4. A list of **computations**, like `[1, 'x', (inc, 'x')]`

So all of the following are valid **computations**:

```
np.array([...])
(add, 1, 2)
(add, 'x', 2)
(add, (inc, 'x'), 2)
(sum, [1, 2])
(sum, ['x', (inc, 'x')])
(np.dot, np.array([...]), np.array([...]))
[(sum, ['x', 'y']), 'z']
```

To encode keyword arguments, we recommend the use of `functools.partial` or `toolz.curry`.

4.9.2 What functions should expect

In cases like `(add, 'x', 'y')`, functions like `add` receive concrete values instead of keys. A dask scheduler replaces keys (like `'x'` and `'y'`) with their computed values (like `1`, and `2`) *before* calling the `add` function.

4.9.3 Entry Point - The `get` function

The `get` function serves as entry point to computation for all schedulers. This function gets the value associated to the given key. That key may refer to stored data, as is the case with `'x'`, or a task as is the case with `'z'`. In the latter case, `get` should perform all necessary computation to retrieve the computed value.

```
>>> from dask.threaded import get

>>> from operator import add

>>> dsk = {'x': 1,
...        'y': 2,
...        'z': (add, 'x', 'y'),
...        'w': (sum, ['x', 'y', 'z'])}
```

```
>>> get(dsk, 'x')
1

>>> get(dsk, 'z')
3

>>> get(dsk, 'w')
6
```

Additionally if given a `list`, get should simultaneously acquire values for multiple keys:

```
>>> get(dsk, ['x', 'y', 'z'])
[1, 2, 3]
```

Because we accept lists of keys as keys, we support nested lists.

```
>>> get(dsk, [['x', 'y'], ['z', 'w']])
[[1, 2], [3, 6]]
```

Internally `get` can be arbitrarily complex, calling out to distributed computing, using caches, and so on.

4.9.4 Why use tuples

With `(add, 'x', 'y')` we wish to encode "the result of calling `add` on the values corresponding to the keys `'x'` and `'y'`.

We intend the following meaning:

```
add('x', 'y')  # after x and y have been replaced
```

But this will err because Python executes the function immediately, before we know values for `'x'` and `'y'`.

We delay the execution by moving the opening parenthesis one term to the left, creating a tuple:

```
Before: add( 'x', 'y')
After: (add, 'x', 'y')
```

This lets us store the desired computation as data that we can analyze using other Python code, rather than cause immediate execution.

LISP users will identify this as an s-expression, or as a rudimentary form of quoting.

4.10 Custom Graphs

There may be times that you want to do parallel computing, but your application doesn't fit neatly into something like `dask.array` or `dask.bag`. In these cases, you can interact directly with the dask schedulers. These schedulers operate well as standalone modules.

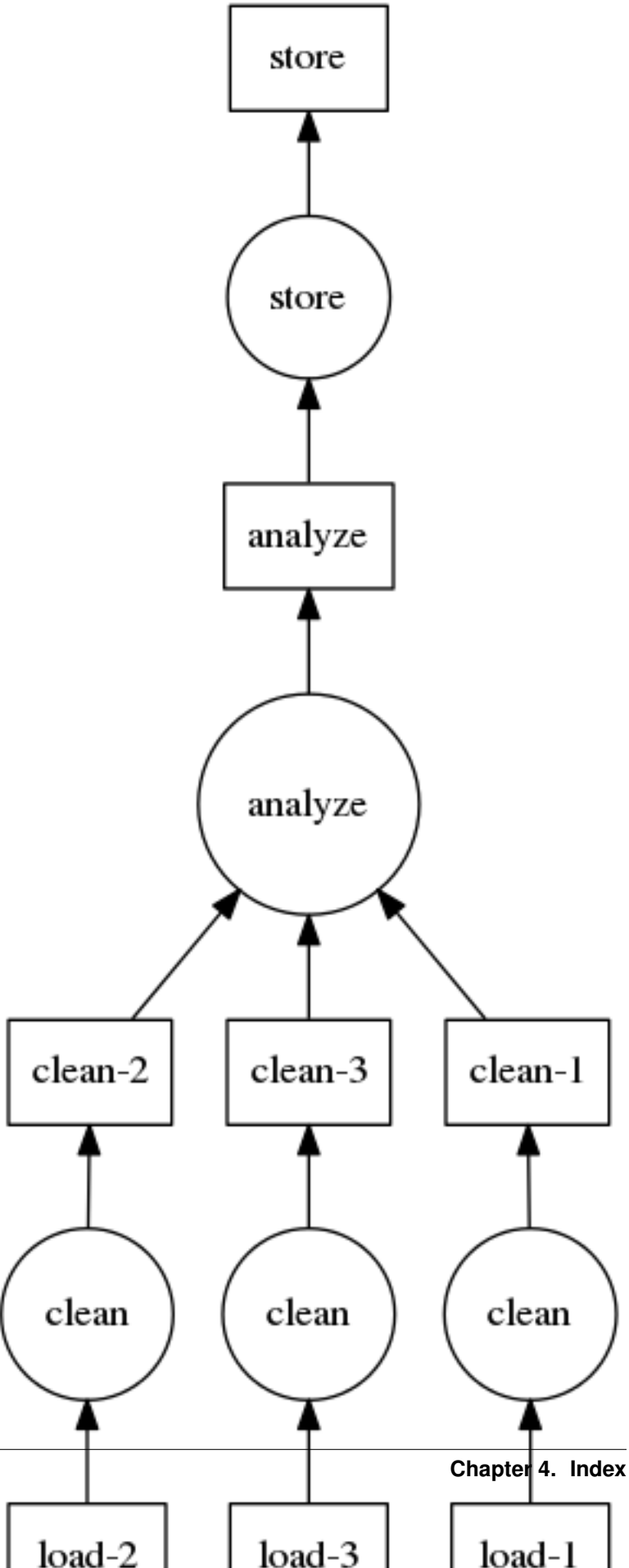
This separation provides a release valve for complex situations and allows advanced projects additional opportunities for parallel execution, even if those projects have an internal representation for their computations. As dask schedulers improve or expand to distributed memory, code written to use dask schedulers will advance as well.

4.10.1 Example

As discussed in the motivation and specification sections, the schedulers take a task graph which is a dict of tuples of functions, and a list of desired keys from that graph.

Here is a mocked out example building a graph for a traditional clean and analyze pipeline:

```
def load(filename):
    ...

def clean(data):
    ...

def analyze(sequence_of_data):
    ...

def store(result):
    with open(..., 'w') as f:
        f.write(result)

dsk = {'load-1': (load, 'myfile.a.data'),
       'load-2': (load, 'myfile.b.data'),
       'load-3': (load, 'myfile.c.data'),
       'clean-1': (clean, 'load-1'),
       'clean-2': (clean, 'load-2'),
       'clean-3': (clean, 'load-3'),
       'analyze': (analyze, ['clean-%d' % i for i in [1, 2, 3]]),
       'store': (store, 'analyze')}

from dask.multiprocessing import get
get(dsk, 'store')  # executes in parallel
```


store
store
analyze
analyze
clean-2
clean-3
clean-1
clean
clean
clean
load-2
load-3
load-1

4.10.2 Related Projects

The following excellent projects also provide parallel execution:

- Joblib
- Multiprocessing
- IPython Parallel
- Concurrent.futures
- Luigi

Each library lets you dictate how your tasks relate to each other with various levels of sophistication. Each library executes those tasks with some internal logic.

Dask schedulers differ in the following ways:

1. You specify the entire graph as a Python dict rather than using a specialized API
2. You get a variety of schedulers ranging from single machine single core, to threaded, to multiprocessing, to distributed, and
3. The dask single-machine schedulers have logic to execute the graph in a way that minimizes memory footprint.

But the other projects offer different advantages and different programming paradigms. One should inspect all such projects before selecting one.

4.11 Optimization

Performance can be significantly improved in different contexts by making small optimizations on the dask graph before calling the scheduler.

The `dask.optimize` module contains several functions to transform graphs in a variety of useful ways. In most cases, users won't need to interact with these functions directly, as specialized subsets of these transforms are done automatically in the dask collections (`dask.array`, `dask.bag`, and `dask.dataframe`). However, users working with custom graphs or computations may find that applying these methods results in substantial speedups.

In general, there are two goals when doing graph optimizations:

1. Simplify computation
2. Improve parallelism.

Simplifying computation can be done on a graph level by removing unnecessary tasks (`cull`), or on a task level by replacing expensive operations with cheaper ones (`RewriteRule`).

Parallelism can be improved by reducing inter-task communication, whether by fusing many tasks into one (`fuse`), or by inlining cheap operations (`inline`, `inline_functions`).

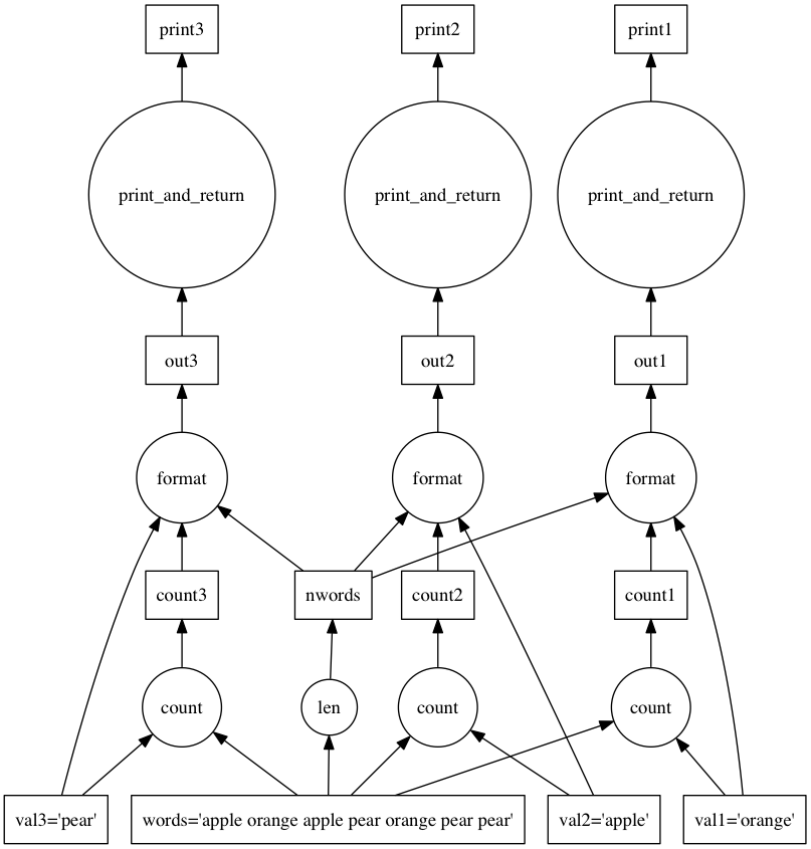
Below, we show an example walking through the use of some of these to optimize a task graph.

4.11.1 Example

Suppose you had a custom dask graph for doing a word counting task:

```
>>> from __future__ import print_function

>>> def print_and_return(string):
...     print(string)
```

```
...     return string

>>> def format_str(count, val, nwords):
...     return ('word list has {0} occurrences of {1}, '
...             'out of {2} words').format(count, val, nwords)

>>> dsk = {'words': 'apple orange apple pear orange pear pear',
...        'nwords': (len, (str.split, 'words')),
...        'val1': 'orange',
...        'val2': 'apple',
...        'val3': 'pear',
...        'count1': (str.count, 'words', 'val1'),
...        'count2': (str.count, 'words', 'val2'),
...        'count3': (str.count, 'words', 'val3'),
...        'out1': (format_str, 'count1', 'val1', 'nwords'),
...        'out2': (format_str, 'count2', 'val2', 'nwords'),
...        'out3': (format_str, 'count3', 'val3', 'nwords'),
...        'print1': (print_and_return, 'out1'),
...        'print2': (print_and_return, 'out2'),
...        'print3': (print_and_return, 'out3')}
```



Here we're counting the occurence of the words `'orange`, `'apple'`, and `'pear'` in the list of words, formatting an output string reporting the results, printing the output, then returning the output string.

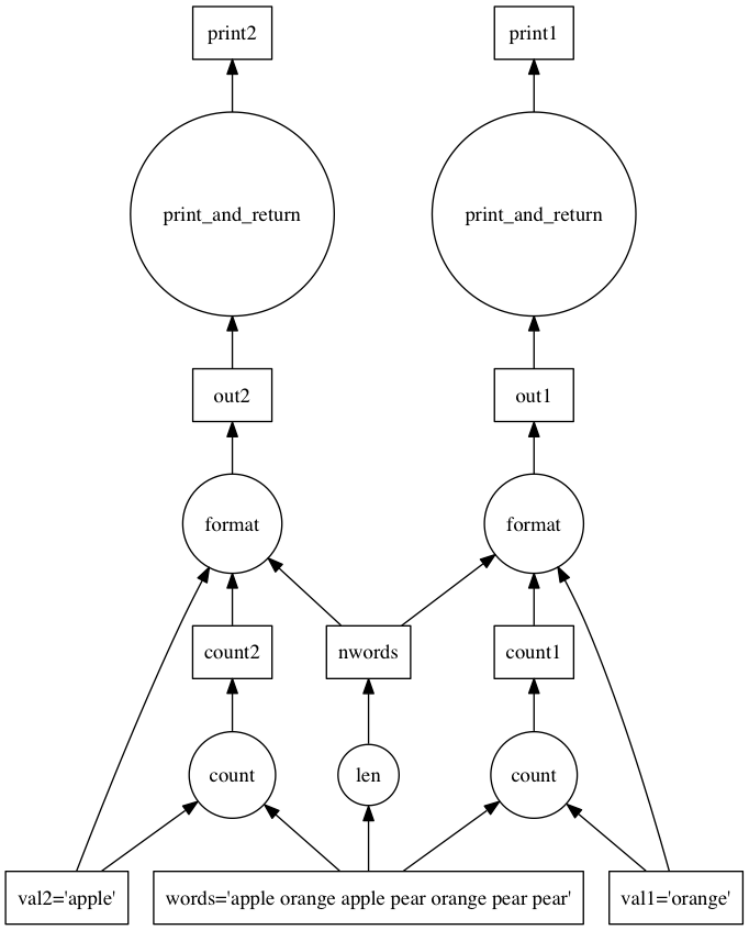
To perform the computation, we pass the dask graph and the desired output keys to a scheduler `get` function:

```
>>> from dask.threaded import get

>>> results = get(dsk, ['print1', 'print2'])
```

```
word list has 2 occurrences of apple, out of 7 words
word list has 2 occurrences of orange, out of 7 words

>>> results
('word list has 2 occurrences of orange, out of 7 words',
 'word list has 2 occurrences of apple, out of 7 words')
```

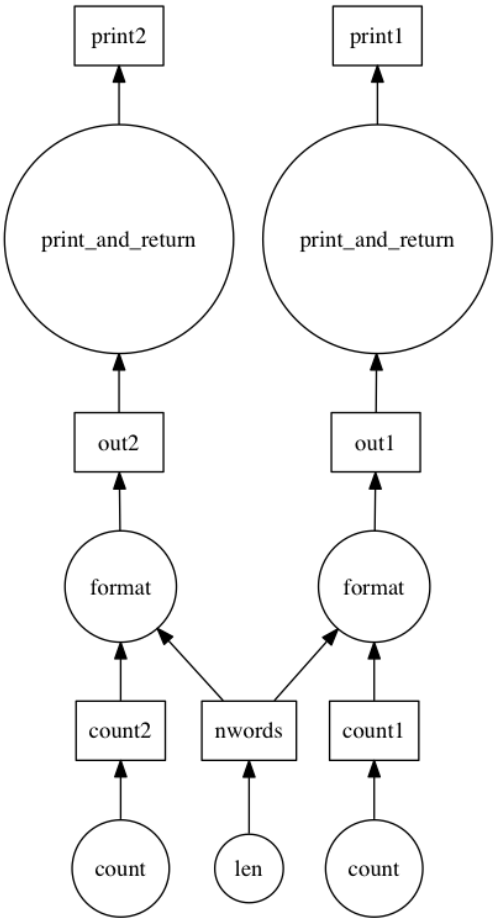
As can be seen above, the scheduler computed only the requested outputs (`'print3'` was never computed). This is because the scheduler internally calls `cull`, which removes the unnecessary tasks from the graph. Even though this is done internally in the scheduler, it can be beneficial to call it at the start of a series of optimizations to reduce the amount of work done in later steps:

```
>>> from dask.optimize import cull
>>> dsk1, dependencies = cull(dsk, ['print1', 'print2'])
```



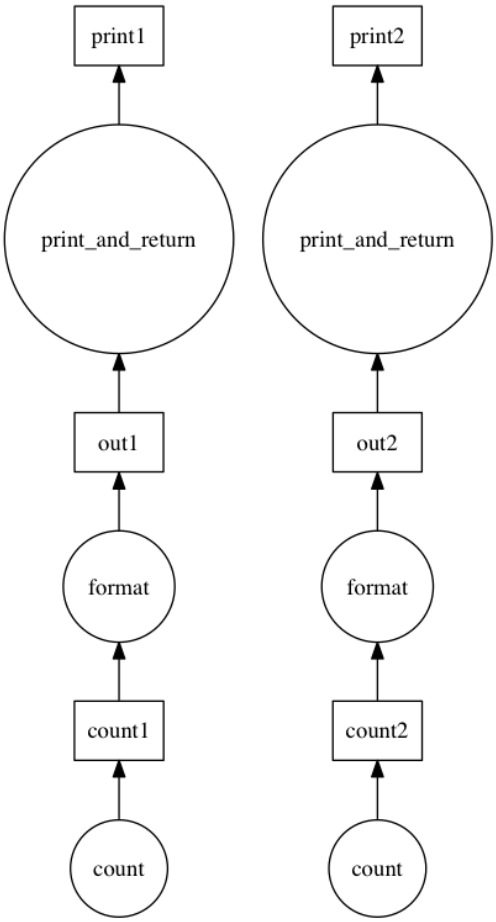
Looking at the task graph above, there are multiple accesses to constants such as `'val1'` or `'val2'` in the dask graph. These can be inlined into the tasks to improve efficiency using the `inline` function. For example:

```
>>> from dask.optimize import inline
>>> dsk2 = inline(dsk1, dependencies=dependencies)
>>> results = get(dsk2, ['print1', 'print2'])
word list has 2 occurrences of apple, out of 7 words
word list has 2 occurrences of orange, out of 7 words
```



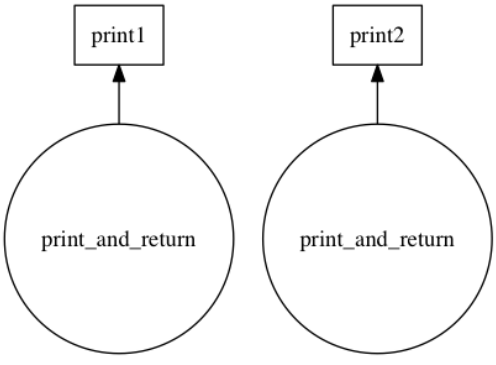
Now we have two sets of *almost* linear task chains. The only link between them is the word counting function. For cheap operations like this, the serialization cost may be larger than the actual computation, so it may be faster to do the computation more than once, rather than passing the results to all nodes. To perform this function inlining, the `inline_functions` function can be used:

```
>>> from dask.optimize import inline_functions
>>> dsk3 = inline_functions(dsk2, [len, str.split], dependencies=dependencies)
>>> results = get(dsk3, ['print1', 'print2'])
word list has 2 occurrences of apple, out of 7 words
word list has 2 occurrences of orange, out of 7 words
```



Now we have a set of purely linear tasks. We'd like to have the scheduler run all of these on the same worker to reduce data serialization between workers. One option is just to merge these linear chains into one big task using the `fuse` function:

```
>>> from dask.optimize import fuse
>>> dsk4, dependencies = fuse(dsk3)
>>> results = get(dsk4, ['print1', 'print2'])
word list has 2 occurrences of apple, out of 7 words
word list has 2 occurrences of orange, out of 7 words
```



Putting it all together:

```
>>> def optimize_and_get(dsk, keys):
...     dsk1, deps = cull(dsk, keys)
...     dsk2 = inline(dsk1, dependencies=deps)
...     dsk3 = inline_functions(dsk2, [len, str.split], dependencies=deps)
...     dsk4, deps = fuse(dsk2)
...     return get(dsk4, keys)

>>> optimize_and_get(dsk, ['print1', 'print2'])
word list has 2 occurrences of apple, out of 7 words
word list has 2 occurrences of orange, out of 7 words
```

In summary, the above operations accomplish the following:

1. Removed tasks unncessary for the desired output using `cull`.
2. Inlined constants using `inline`.
3. Inlined cheap computations using `inline_functions`, improving parallelism.
4. Fused linear tasks together to ensure they run on the same worker using `fuse`.

As stated previously, these optimizations are already performed automatically in the dask collections. Users not working with custom graphs or computations should rarely need to directly interact with them.

These are just a few of the optimizations provided in `dask.optimize`. For more information, see the API below.

4.11.2 Rewrite Rules

For context based optimizations, `dask.rewrite` provides functionality for pattern matching and term rewriting. This is useful for replacing expensive computations with equivalent, cheaper computations. For example, `dask.array` uses the rewrite functionality to replace series of array slicing operations with a more efficient single slice.

The interface to the rewrite system consists of two classes:

1. `RewriteRule(lhs, rhs, vars)`

   Given a left-hand-side (`lhs`), a right-hand-side (`rhs`), and a set of variables (`vars`), a rewrite rule declaratively encodes the following operation:

   ```
   lhs -> rhs if task matches lhs over variables
   ```

2. `RuleSet(*rules)`

   A collection of rewrite rules. The design of `RuleSet` class allows for efficient "many-to-one" pattern matching, meaning that there is minimal overhead for rewriting with multiple rules in a rule set.

Example

Here we create two rewrite rules expressing the following mathematical transformations:

1. `a + a -> 2*a`
2. `a * a -> a**2`

where `'a'` is a variable:

```
>>> from dask.rewrite import RewriteRule, RuleSet
>>> from operator import add, mul, pow

>>> variables = ('a',)

>>> rule1 = RewriteRule((add, 'a', 'a'), (mul, 'a', 2), variables)

>>> rule2 = RewriteRule((mul, 'a', 'a'), (pow, 'a', 2), variables)

>>> rs = RuleSet(rule1, rule2)
```

The `RewriteRule` objects describe the desired transformations in a declarative way, and the `RuleSet` builds an efficient automata for applying that transformation. Rewriting can then be done using the `rewrite` method:

```
>>> rs.rewrite((add, 5, 5))
(mul, 5, 2)

>>> rs.rewrite((mul, 5, 5))
(pow, 5, 2)

>>> rs.rewrite((mul, (add, 3, 3), (add, 3, 3)))
(pow, (mul, 3, 2), 2)
```

The whole task is traversed by default. If you only want to apply a transform to the top-level of the task, you can pass in `strategy='top_level'` as shown:

```
# Transforms whole task
>>> rs.rewrite((sum, [(add, 3, 3), (mul, 3, 3)]))
(sum, [(mul, 3, 2), (pow, 3, 2)])

# Only applies to top level, no transform occurs
>>> rs.rewrite((sum, [(add, 3, 3), (mul, 3, 3)]), strategy='top_level')
(sum, [(add, 3, 3), (mul, 3, 3)])
```

The rewriting system provides a powerful abstraction for transforming computations at a task level. Again, for many users, directly interacting with these transformations will be unnecessary.

4.11.3 API

Top level optimizations

cull(dsk, keys)	Return new dask with only the tasks required to calculate keys.
fuse(dsk[, keys, dependencies])	Return new dask graph with linear sequence of tasks fused together.
inline(dsk[, keys, inline_constants, ...])	Return new dask with the given keys inlined with their values.
inline_functions(dsk, output[, ...])	Inline cheap functions into larger operations

Utility functions

dealias(dsk[, keys])	Remove aliases from dask
dependency_dict(dsk)	Create a dict matching ordered dependencies to keys.
equivalent(term1, term2[, subs])	Determine if two terms are equivalent, modulo variable substitution.
functions_of(task)	Set of functions contained within nested task
merge_sync(dsk1, dsk2)	Merge two dasks together, combining equivalent tasks.
sync_keys(dsk1, dsk2)	Return a dict matching keys in *dsk2* to equivalent keys in *dsk1*.

Rewrite Rules

RewriteRule(lhs, rhs[, vars])	A rewrite rule.
RuleSet(*rules)	A set of rewrite rules.

Definitions

dask.optimize.**cull**(*dsk*, *keys*)

Return new dask with only the tasks required to calculate keys.

In other words, remove unnecessary tasks from dask. `keys` may be a single key or list of keys.

Returns dsk: culled dask graph

dependencies: Dict mapping {key: [deps]}. Useful side effect to accelerate

other optimizations, notably fuse.

Examples

```
>>> d = {'x': 1, 'y': (inc, 'x'), 'out': (add, 'x', 10)}
>>> dsk, dependencies = cull(d, 'out')
>>> dsk
{'x': 1, 'out': (add, 'x', 10)}
>>> dependencies
{'x': set(), 'out': set(['x'])}
```

dask.optimize.**fuse**(*dsk*, *keys=None*, *dependencies=None*)

Return new dask graph with linear sequence of tasks fused together.

If specified, the keys in `keys` keyword argument are *not* fused. Supply `dependencies` from output of `cull` if available to avoid recomputing dependencies.

Parameters **dsk: dict**

keys: list

dependencies: dict, optional

{key: [list-of-keys]}. Must be a list to provide count of each key This optional input often comes from `cull`

Returns dsk: output graph with keys fused

dependencies: dict mapping dependencies after fusion. Useful side effect

to accelerate other downstream optimizations.

Examples

```
>>> d = {'a': 1, 'b': (inc, 'a'), 'c': (inc, 'b')}
>>> dsk, dependencies = fuse(d)
>>> dsk
{'c': (inc, (inc, 1))}
>>> dsk, dependencies = fuse(d, keys=['b'])
>>> dsk
{'b': (inc, 1), 'c': (inc, 'b')}
```

dask.optimize.**inline**(*dsk*, *keys=None*, *inline_constants=True*, *dependencies=None*)
Return new dask with the given keys inlined with their values.

Inlines all constants if `inline_constants` keyword is True.

Examples

```
>>> d = {'x': 1, 'y': (inc, 'x'), 'z': (add, 'x', 'y')}
>>> inline(d)
{'y': (inc, 1), 'z': (add, 1, 'y')}
```

```
>>> inline(d, keys='y')
{'z': (add, 1, (inc, 1))}
```

```
>>> inline(d, keys='y', inline_constants=False)
{'x': 1, 'z': (add, 'x', (inc, 'x'))}
```

dask.optimize.**inline_functions**(*dsk*, *output*, *fast_functions=None*, *inline_constants=False*, *dependencies=None*)
Inline cheap functions into larger operations

Examples

```
>>> dsk = {'out': (add, 'i', 'd'),
...        'i': (inc, 'x'),
...        'd': (double, 'y'),
...        'x': 1, 'y': 1}
>>> inline_functions(dsk, [], [inc])
{'out': (add, (inc, 'x'), 'd'),
 'd': (double, 'y'),
 'x': 1, 'y': 1}
```

Protect output keys. In the example below `i` is not inlined because it is marked as an output key.

```
>>> inline_functions(dsk, ['i', 'out'], [inc, double])
{'out': (add, 'i', (double, 'y')),
 'i': (inc, 'x'),
 'x': 1, 'y': 1}
```

dask.optimize.**dealias**(*dsk*, *keys=None*)
Remove aliases from dask

Removes and renames aliases using `inline`. Keeps aliases at the top of the DAG to ensure entry points stay the same.

Aliases are not expected by schedulers. It's unclear that this is a legal state.

Optional `keys` keyword argument allows us to protect keys from being deleted. This is useful to protect keys that would be expected by a scheduler.

Examples

```
>>> dsk = {'a': (range, 5),
...        'b': 'a',
...        'c': 'b',
...        'd': (sum, 'c'),
```

```
...         'e': 'd',
...         'f': (inc, 'd')}
```

```
>>> dealias(dsk)
{'a': (range, 5),
 'd': (sum, 'a'),
 'e': (identity, 'd'),
 'f': (inc, 'd')}
```

```
>>> dsk = {'a': (range, 5),
...         'b': 'a'}
>>> dealias(dsk)
{'b': (range, 5)}
```

```
>>> dealias(dsk, keys=['a', 'b'])
{'a': (range, 5),
 'b': (identity, 5)}
```

dask.optimize.**dependency_dict**(*dsk*)

Create a dict matching ordered dependencies to keys.

Examples

```
>>> from operator import add
>>> dsk = {'a': 1, 'b': 2, 'c': (add, 'a', 'a'), 'd': (add, 'b', 'a')}
>>> dependency_dict(dsk)
{(): ['a', 'b'], ('a', 'a'): ['c'], ('b', 'a'): ['d']}
```

dask.optimize.**equivalent**(*term1*, *term2*, *subs=None*)

Determine if two terms are equivalent, modulo variable substitution.

Equivalent to applying substitutions in *subs* to *term2*, then checking if *term1* == *term2*.

If a subterm doesn't support comparison (i.e. *term1* == *term2* errors), returns *False*.

Parameters **term1, term2** : terms

subs : dict, optional

Mapping of substitutions from *term2* to *term1*

Examples

```
>>> from operator import add
>>> term1 = (add, 'a', 'b')
>>> term2 = (add, 'x', 'y')
>>> subs = {'x': 'a', 'y': 'b'}
>>> equivalent(term1, term2, subs)
True
>>> subs = {'x': 'a'}
>>> equivalent(term1, term2, subs)
False
```

dask.optimize.**functions_of**(*task*)

Set of functions contained within nested task

Examples

```
>>> task = (add, (mul, 1, 2), (inc, 3))
>>> functions_of(task)
set([add, mul, inc])
```

dask.optimize.**merge_sync**(*dsk1*, *dsk2*)

Merge two dasks together, combining equivalent tasks.

If a task in *dsk2* exists in *dsk1*, the task and key from *dsk1* is used. If a task in *dsk2* has the same key as a task in *dsk1* (and they aren't equivalent tasks), then a new key is created for the task in *dsk2*. This prevents name conflicts.

Parameters **dsk1, dsk2** : dict

Variable names in *dsk2* are replaced with equivalent ones in *dsk1* before merging.

Returns **new_dsk** : dict

The merged dask.

key_map : dict

A mapping between the keys from *dsk2* to their new names in *new_dsk*.

Examples

```
>>> from operator import add, mul
>>> dsk1 = {'a': 1, 'b': (add, 'a', 10), 'c': (mul, 'b', 5)}
>>> dsk2 = {'x': 1, 'y': (add, 'x', 10), 'z': (mul, 'y', 2)}
>>> new_dsk, key_map = merge_sync(dsk1, dsk2)
>>> new_dsk
{'a': 1, 'b': (add, 'a', 10), 'c': (mul, 'b', 5), 'z': (mul, 'b', 2)}
>>> key_map
{'x': 'a', 'y': 'b', 'z': 'z'}
```

Conflicting names are replaced with auto-generated names upon merging.

```
>>> dsk1 = {'a': 1, 'res': (add, 'a', 1)}
>>> dsk2 = {'x': 1, 'res': (add, 'x', 2)}
>>> new_dsk, key_map = merge_sync(dsk1, dsk2)
>>> new_dsk
{'a': 1, 'res': (add, 'a', 1), 'merge_1': (add, 'a', 2)}
>>> key_map
{'x': 'a', 'res': 'merge_1'}
```

dask.optimize.**sync_keys**(*dsk1*, *dsk2*)

Return a dict matching keys in *dsk2* to equivalent keys in *dsk1*.

Parameters **dsk1, dsk2** : dict

Examples

```
>>> from operator import add, mul
>>> dsk1 = {'a': 1, 'b': (add, 'a', 10), 'c': (mul, 'b', 5)}
>>> dsk2 = {'x': 1, 'y': (add, 'x', 10), 'z': (mul, 'y', 2)}
>>> sync_keys(dsk1, dsk2)
{'x': 'a', 'y': 'b'}
```

dask.rewrite.**RewriteRule**(*lhs*, *rhs*, *vars=()*)

A rewrite rule.

Expresses *lhs* -> *rhs*, for variables *vars*.

Parameters **lhs** : task

The left-hand-side of the rewrite rule.

rhs : task or function

The right-hand-side of the rewrite rule. If it's a task, variables in *rhs* will be replaced by terms in the subject that match the variables in *lhs*. If it's a function, the function will be called with a dict of such matches.

vars: tuple, optional

Tuple of variables found in the lhs. Variables can be represented as any hashable object; a good convention is to use strings. If there are no variables, this can be omitted.

Examples

Here's a *RewriteRule* to replace all nested calls to *list*, so that *(list, (list, 'x'))* is replaced with *(list, 'x')*, where *'x'* is a variable.

```
>>> lhs = (list, (list, 'x'))
>>> rhs = (list, 'x')
>>> variables = ('x',)
>>> rule = RewriteRule(lhs, rhs, variables)
```

Here's a more complicated rule that uses a callable right-hand-side. A callable *rhs* takes in a dictionary mapping variables to their matching values. This rule replaces all occurrences of *(list, 'x')* with *'x'* if *'x'* is a list itself.

```
>>> lhs = (list, 'x')
>>> def repl_list(sd):
...     x = sd['x']
...     if isinstance(x, list):
...         return x
...     else:
...         return (list, x)
>>> rule = RewriteRule(lhs, repl_list, variables)
```

dask.rewrite.**RuleSet**(**rules*)

A set of rewrite rules.

Forms a structure for fast rewriting over a set of rewrite rules. This allows for syntactic matching of terms to patterns for many patterns at the same time.

Examples

```
>>> def f(*args): pass
>>> def g(*args): pass
>>> def h(*args): pass
>>> from operator import add
```

```
>>> rs = RuleSet(                  # Make RuleSet with two Rules
...         RewriteRule((add, 'x', 0), 'x', ('x',)),
...         RewriteRule((f, (g, 'x'), 'y'),
```

```
...                         (h, 'x', 'y'),
...                         ('x', 'y')))
```

```
>>> rs.rewrite((add, 2, 0))        # Apply ruleset to single task
2
```

```
>>> rs.rewrite((f, (g, 'a', 3)))
(h, 'a', 3)
```

```
>>> dsk = {'a': (add, 2, 0),       # Apply ruleset to full dask graph
...        'b': (f, (g, 'a', 3))}
```

```
>>> from toolz import valmap
>>> valmap(rs.rewrite, dsk)
{'a': 2,
 'b': (h, 'a', 3)}
```

Attributes

rules	(list) A list of *RewriteRule*'s included in the 'RuleSet.

Scheduling

Schedulers execute task graphs. Dask currently has two main schedulers, one for single machine processing using threads or processes, and one for distributed memory clusters.

- Scheduler Overview
- Single machine scheduler
- Distributed scheduler (separate webpage)
- Scheduling in Depth

4.12 Scheduler Overview

After we create a dask graph, we use a scheduler to run it. Dask currently implements a few different schedulers:

- `dask.threaded.get`: a scheduler backed by a thread pool
- `dask.multiprocessing.get`: a scheduler backed by a process pool
- `dask.async.get_sync`: a synchronous scheduler, good for debugging
- **`distributed.Executor.get`: a distributed scheduler for executing graphs** on multiple machines. This lives in the external distributed project.

4.12.1 The `get` function

The entry point for all schedulers is a `get` function. This takes a dask graph, and a key or list of keys to compute:

```
>>> dsk = {'a': 1,
...        'b': 2,
...        'c': (add, 'a', 'b'),
...        'd': (sum, ['a', 'b', 'c'])}
```

```
>>> get(dsk, 'c')
3

>>> get(dsk, 'd')
6

>>> get(dsk, ['a', 'b', 'c'])
[1, 2, 3]
```

4.12.2 Using `compute` methods

When working with dask collections, you will rarely need to interact with scheduler `get` functions directly. Each collection has a default scheduler, and a built-in `compute` method that calculates the output of the collection:

```
>>> import dask.array as da
>>> x = da.arange(100, chunks=10)
>>> x.sum().compute()
4950
```

The compute method takes a number of keywords:

- `get`: a scheduler `get` function, overrides the default for the collection
- `**kwargs`: extra keywords to pass on to the scheduler `get` function.

See also: *Configuring the schedulers*.

4.12.3 The `compute` function

You may wish to compute results from multiple dask collections at once. Similar to the `compute` method on each collection, there is a general `compute` function that takes multiple collections and returns multiple results. This merges the graphs from each collection, so intermediate results are shared:

```
>>> y = (x + 1).sum()
>>> z = (x + 1).mean()
>>> da.compute(y, z)     # Compute y and z, sharing intermediate results
(5050, 50.5)
```

Here the `x + 1` intermediate was only computed once, while calling `y.compute()` and `z.compute()` would compute it twice. For large graphs that share many intermediates, this can be a big performance gain.

The `compute` function works with any dask collection, and is found in `dask.base`. For convenience it has also been imported into the top level namespace of each collection.

```
>>> from dask.base import compute
>>> compute is da.compute
True
```

4.12.4 Configuring the schedulers

The dask collections each have a default scheduler:

- `dask.array` and `dask.dataframe` use the threaded scheduler by default
- `dask.bag` uses the multiprocessing scheduler by default.

For most cases, the default settings are good choices. However, sometimes you may want to use a different scheduler. There are two ways to do this.

1. Using the `get` keyword in the `compute` method:

```
>>> x.sum().compute(get=dask.multiprocessing.get)
```

2. Using `set_options`. This can be used either as a context manager, or to set the scheduler globally:

```
# As a context manager
>>> with set_options(get=dask.multiprocessing.get):
...     x.sum().compute()

# Set globally
>>> set_options(get=dask.multiprocessing.get)
>>> x.sum().compute()
```

Additionally, each scheduler may take a few extra keywords specific to that scheduler. For example, the multiprocessing and threaded schedulers each take a `num_workers` keyword, which sets the number of processes or threads to use (defaults to number of cores). This can be set by passing the keyword when calling `compute`:

```
# Compute with 4 threads
>>> x.compute(num_workers=4)
```

Alternatively, the multiprocessing and threaded schedulers will check for a global pool set with `set_options`:

```
>>> from multiprocessing.pool import ThreadPool
>>> with set_options(pool=ThreadPool(4)):
...     x.compute()
```

For more information on the individual options for each scheduler, see the docstrings for each scheduler `get` function.

4.12.5 Debugging the schedulers

Debugging parallel code can be difficult, as conventional tools such as `pdb` don't work well with multiple threads or processes. To get around this when debugging, we recommend using the synchronous scheduler found at `dask.async.get_sync`. This runs everything serially, allowing it to work well with `pdb`:

```
>>> set_options(get=dask.async.get_sync)
>>> x.sum().compute()     # This computation runs serially instead of in parallel
```

The shared memory schedulers also provide a set of callbacks that can be used for diagnosing and profiling. You can learn more about scheduler callbacks and diagnostics here.

4.12.6 More Information

- See Shared Memory for information on the design of the shared memory (threaded or multiprocessing) schedulers
- See distributed for information on the distributed memory scheduler

4.13 Shared Memory

The asynchronous scheduler requires an `apply_async` function and a `Queue`. These determine the kind of worker and parallelism that we exploit. `apply_async` functions can be found in the following places:

- `multithreading.Pool().apply_async` - uses multiple processes
- `multithreading.pool.ThreadPool().apply_async` - uses multiple threads
- `dask.async.apply_sync` - uses only the main thread (useful for debugging)

Full dask `get` functions exist in each of `dask.threaded.get`, `dask.multiprocessing.get` and `dask.async.get_sync` respectively.

4.13.1 Policy

The asynchronous scheduler maintains indexed data structures that show which tasks depend on which data, what data is available, and what data is waiting on what tasks to complete before it can be released, and what tasks are currently running. It can update these in constant time relative to the number of total and available tasks. These indexed structures make the dask async scheduler scalable to very many tasks on a single machine.

To keep the memory footprint small, we choose to keep ready-to-run tasks in a LIFO stack such that the most recently made available tasks get priority. This encourages the completion of chains of related tasks before new chains are started. This can also be queried in constant time.

More info: scheduling policy.

4.13.2 Performance

tl;dr The threaded scheduler overhead behaves roughly as follows:

- 1ms overhead per task
- 100ms startup time (if you wish to make a new ThreadPool each time)
- Constant scaling with number of tasks
- Linear scaling with number of dependencies per task

Schedulers introduce overhead. This overhead effectively limits the granularity of our parallelism. Below we measure overhead of the async scheduler with different apply functions (threaded, sync, multiprocessing), and under different kinds of load (embarrassingly parallel, dense communication).

The quickest/simplest test we can do it to use IPython's `timeit` magic:

```
In [1]: import dask.array as da

In [2]: x = da.ones(1000, chunks=(2,)).sum()

In [3]: len(x.dask)
Out[3]: 1001

In [4]: %timeit x.compute()
1 loops, best of 3: 550 ms per loop
```

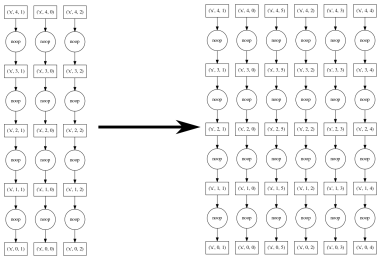
So this takes about 500 microseconds per task. About 100ms of this is from overhead:

```
In [6]: x = da.ones(1000, chunks=(1000,)).sum()
In [7]: %timeit x.compute()
10 loops, best of 3: 103 ms per loop
```

Most of this overhead is from spinning up a ThreadPool each time. This may be mediated by using a global or contextual pool:

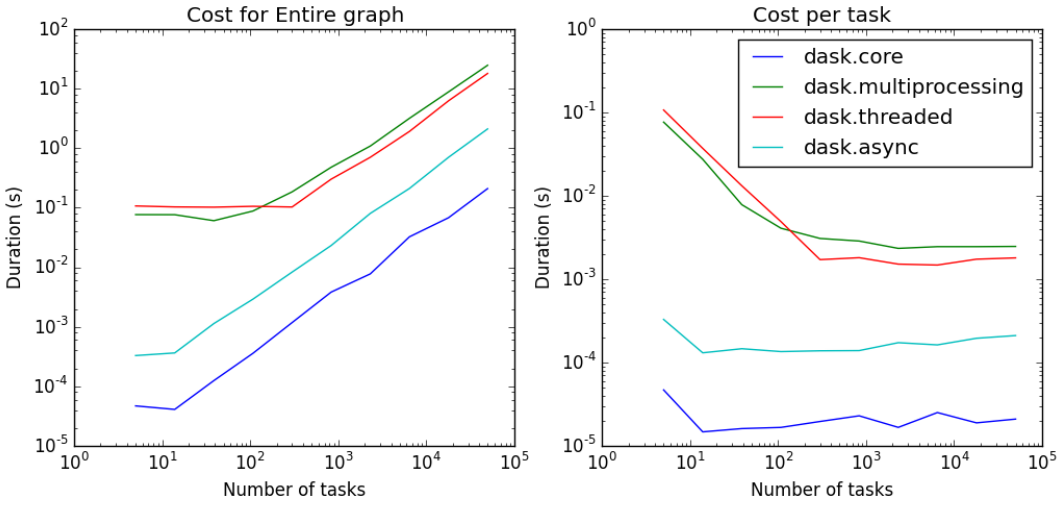
```
>>> from multiprocessing.pool import ThreadPool
>>> pool = ThreadPool()
>>> da.set_options(pool=pool)   # set global threadpool

or

>>> with set_options(pool=pool)   # use threadpool throughout with block
...      ...
```

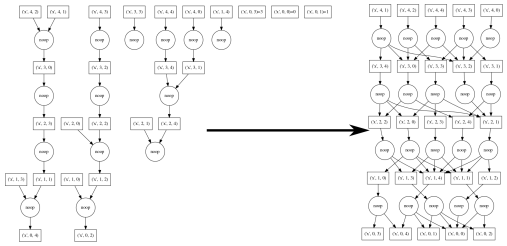
We now measure scaling the number of tasks and scaling the density of the graph:



Linear scaling with number of tasks

As we increase the number of tasks in a graph, we see that the scheduling overhead grows linearly. The asymptotic cost per task depends on the scheduler. The schedulers that depend on some sort of asynchronous pool have costs of a few milliseconds and the single threaded schedulers have costs of a few microseconds.

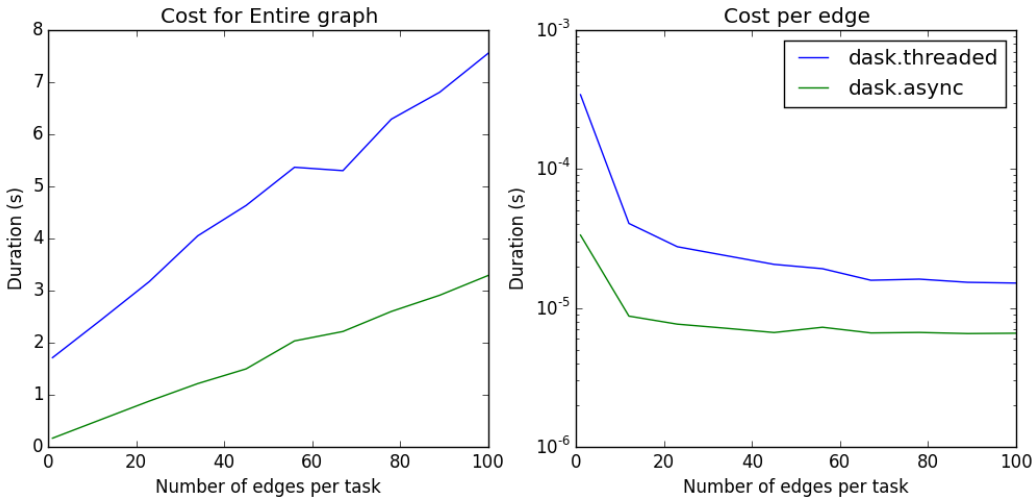




Linear scaling with number of edges

As we increase the number of edges per task, the scheduling overhead again increases linearly.

Note: Neither the naive core scheduler nor the multiprocessing scheduler are good at workflows with non-trivial cross-task communication; they have been removed from the plot.



Download scheduling script

4.13.3 Known Limitations

The shared memory scheduler has some notable limitations:

1. It works on a single machine
2. The threaded scheduler is limited by the GIL on Python code, so if your operations are pure python functions, you should not expect a multi-core speedup
3. The multiprocessing scheduler must serialize functions between workers, which can fail
4. The multiprocessing scheduler must serialize data between workers and the central process, which can be expensive
5. The multiprocessing scheduler cannot transfer data directly between worker processes; all data routes through the master process.

4.14 Scheduling in Depth

Note: this technical document is not optimized for user readability.

The default shared memory scheduler used by most dask collections lives in `dask/async.py`. This scheduler dynamically schedules tasks to new workers as they become available. It operates in a shared memory environment without consideration to data locality, all workers have access to all data equally.

We find that our workloads are best served by trying to minimize the memory footprint. This document talks about our policies to accomplish this in our scheduling budget of one millisecond per task, irrespective of the number of tasks.

Generally we are faced with the following situation: A worker arrives with a newly completed task. We update our data structures of execution state and have to provide a new task for that worker. In general there are very many available tasks, which should we give to the worker?

Q: Which of our available tasks should we give to the newly ready worker?

This question is simple and local and yet strongly impacts the performance of our algorithm. We want to choose a task that lets us free memory now and in the future. We need a clever and cheap way to break a tie between the set of available tasks.

At this stage we choose the policy of "last in, first out." That is we choose the task that was most recently made available, quite possibly by the worker that just returned to us. This encourages the general theme of finishing things before starting new things.

We implement this with a stack. When a worker arrives with its finished task we figure out what new tasks we can now compute with the new data and put those on top of the stack if any exist. We pop an item off of the top of the stack and deliver that to the waiting worker.

And yet if the newly completed task makes ready multiple newly ready tasks in which order should we place them on the stack? This is yet another opportunity for a tie breaker. This is particularly important at *the beginning* of execution where we typically add a large number of leaf tasks onto the stack. Our choice in this tie breaker also strongly affects performance in many cases.

We want to encourage depth first behavior where, if our computation is composed of something like many trees we want to fully explore one subtree before moving on to the next. This encourages our workers to complete blocks/subtrees of our graph before moving on to new blocks/subtrees.

And so to encourage this "depth first behavior" we do a depth first search and number all nodes according to their number in the depth first search (DFS) traversal. We use this number to break ties when adding tasks on to the stack. Please note that while we spoke of optimizing the many-distinct-subtree case above this choice is entirely local and applies quite generally beyond this case. Anything that behaves even remotely like the many-distinct-subtree case will benefit accordingly, and this case is quite common in normal workloads.

And yet we have glossed over another tie breaker. Performing the depth first search, when we arrive at a node with many children we can choose the order in which to traverse the children. We resolve this tie breaker by selecting those children whose result is depended upon by the most nodes. This dependence can be either direct for those nodes that take that data as input or indirect for any ancestor node in the graph. This emphasizing traversing first those nodes that are parts of critical paths having long vertical chains that rest on top of this node's result, and nodes whose data is depended upon by many nodes in the future. We choose to dive down into these subtrees first in our depth first search so that future computations don't get stuck waiting for them to complete.

And so we have three tie breakers

1. Q: Which of these available tasks should I run?

   A: Last in, first out

2. Q: Which of these tasks should I put on the stack first?

   A: Do a depth first search before the computation, use that ordering.

3. Q: When performing the depth first search how should I choose between children?

   A: Choose those children on whom the most data depends

We have found common workflow types that require each of these decisions. We have not yet run into a commonly occurring graph type in data analysis that is not well handled by these heuristics for the purposes of minimizing memory use.

Inspecting and Diagnosing Graphs

Parallel code can be tricky to debug and profile. Dask provides a few tools to help make debugging and profiling graph execution easier.

- Inspecting Dask objects
- Diagnostics

4.15 Inspecting Dask objects

Dask itself is just a specification on top of normal Python dictionaries. Objects like `dask.Array` are just a thin wrapper around these dictionaries with a little bit of shape metadata.

Users should only have to interact with the higher-level `Array` objects. Developers may want to dive more deeply into the dictionaries/task graphs themselves

4.15.1 `dask` attribute

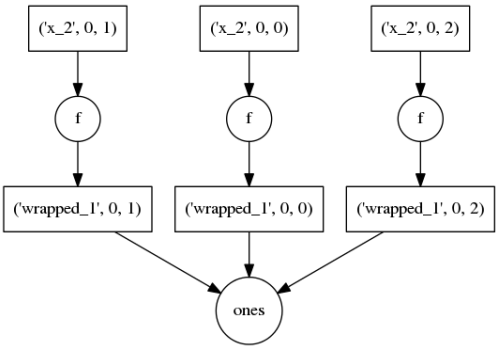
The first step is to look at the `.dask` attribute of an array

```
>>> import dask.array as da
>>> x = da.ones((5, 15), chunks=(5, 5))
>>> x.dask
{('wrapped_1', 0, 0): (ones, (5, 5)),
 ('wrapped_1', 0, 1): (ones, (5, 5)),
 ('wrapped_1', 0, 2): (ones, (5, 5))}
```

This attribute becomes more interesting as you perform operations on your Array objects

```
>>> (x + 1).dask
{('wrapped_1', 0, 0): (ones, (5, 5)),
 ('wrapped_1', 0, 1): (ones, (5, 5)),
 ('wrapped_1', 0, 2): (ones, (5, 5))
 ('x_1', 0, 0): (add, ('wrapped_1', 0, 0), 1),
 ('x_1', 0, 1): (add, ('wrapped_1', 0, 1), 1),
 ('x_1', 0, 2): (add, ('wrapped_1', 0, 2), 1)}
```

4.15.2 Visualize graphs with DOT



If you have basic graphviz tools like `dot` installed then dask can also generate visual graphs from your task graphs.

```
>>> d = (x + 1).dask
>>> from dask.dot import dot_graph
>>> dot_graph(d)
Writing graph to mydask.pdf
```

The result is shown to the right.

4.16 Diagnostics

Profiling parallel code can be tricky, but `dask.diagnostics` provides functionality to aid in profiling and inspecting dask graph execution.

4.16.1 Scheduler Callbacks

Schedulers based on `dask.async.get_async` (currently `dask.async.get_sync`, `dask.threaded.get`, and `dask.multiprocessing.get`) accept four callbacks, allowing for inspection of scheduler execution.

The callbacks are:

1. `start(dsk)`

   Run at the beginning of execution, right before the state is initialized. Receives the dask graph.

2. `start_state(dsk, state)`

   Run at the beginning of execution, right after the state is initialized. Receives the dask graph and scheduler state.

3. `pretask(key, dsk, state)`

   Run every time a new task is started. Receives the key of the task to be run, the dask graph, and the scheduler state.

4. `posttask(key, result, dsk, state, id)`

   Run every time a task is finished. Receives the key of the task that just completed, the result, the dask graph, the scheduler state, and the id of the worker that ran the task.

5. `finish(dsk, state, errored)`

   Run at the end of execution, right before the result is returned. Receives the dask graph, the scheduler state, and a boolean indicating whether or not the exit was due to an error.

These are internally represented as tuples of length 4, stored in the order presented above. Callbacks for common use cases are provided in `dask.diagnostics`.

4.16.2 Progress Bar

The `ProgressBar` class builds on the scheduler callbacks described above to display a progress bar in the terminal or notebook during computation. This can be a nice feedback during long running graph execution. It can be used as a context manager around calls to `get` or `compute` to profile the computation:

```
>>> from dask.diagnostics import ProgressBar
>>> a = da.random.normal(size=(10000, 10000), chunks=(1000, 1000))
>>> res = a.dot(a.T).mean(axis=0)

>>> with ProgressBar():
...     out = res.compute()
[########################################] | 100% Completed | 17.1 s
```

Or registered globally using the `register` method.

```
>>> pbar = ProgressBar()
>>> pbar.register()
>>> out = res.compute()
[########################################] | 100% Completed | 17.1 s
```

To unregister from the global callbacks, call the `unregister` method:

```
>>> pbar.unregister()
```

4.16.3 Profiling

Dask provides a few tools for profiling execution. As with the `ProgressBar`, they each can be used as context managers, or registered globally.

Profiler

The `Profiler` class is used to profile dask execution at the task level. During execution it records the following information for each task:

1. Key
2. Task
3. Start time in seconds since the epoch
4. Finish time in seconds since the epoch
5. Worker id

ResourceProfiler

The `ResourceProfiler` class is used to profile dask execution at the resource level. During execution it records the following information for each timestep

1. Time in seconds since the epoch
2. Memory usage in MB
3. % CPU usage

The default timestep is 1 second, but can be set manually using the `dt` keyword.

```
>>> from dask.diagnostics import ResourceProfiler
>>> rprof = ResourceProfiler(dt=0.5)
```

CacheProfiler

The `CacheProfiler` class is used to profile dask execution at the scheduler cache level. During execution it records the following information for each task:

1. Key
2. Task
3. Size metric
4. Cache entry time in seconds since the epoch
5. Cache exit time in seconds since the epoch

Where the size metric is the output of a function called on the result of each task. The default metric is to count each task (`metric` is 1 for all tasks). Other functions may be used as a metric instead through the `metric` keyword. For example, the `nbytes` function found in `cachey` can be used to measure the number of bytes in the scheduler cache:

```
>>> from dask.diagnostics import CacheProfiler
>>> from cachey import nbytes
>>> cprof = CacheProfiler(metric=nbytes)
```

Example

As an example to demonstrate using the diagnostics, we'll profile some linear algebra done with `dask.array`. We'll create a random array, take its QR decomposition, and then reconstruct the initial array by multiplying the Q and R components together. Note that since the profilers (and all diagnostics) are just context managers, multiple profilers can be used in a with block:

```
>>> import dask.array as da
>>> from dask.diagnostics import Profiler, ResourceProfiler, CacheProfiler
>>> a = da.random.random(size=(10000, 1000), chunks=(1000, 1000))
>>> q, r = da.linalg.qr(a)
>>> a2 = q.dot(r)

>>> with Profiler() as prof, ResourceProfiler(dt=0.25) as rprof,
...         CacheProfiler(metric=nbytes) as cprof:
...     out = a2.compute()
```

The results of each profiler are stored in their `results` attribute as a list of `namedtuple` objects:

```
>>> prof.results[0]
TaskData(key=('tsqr-8d16e396b237bf7a731333130d310cb9_QR_st1', 5, 0),
         task=(qr, (_apply_random, 'random_sample', 1060164455, (1000, 1000), (), {})),
         start_time=1454368444.493292,
         end_time=1454368444.902987,
         worker_id=4466937856)

>>> rprof.results[0]
ResourceData(time=1454368444.078748, mem=74.100736, cpu=0.0)

>>> cprof.results[0]
CacheData(key=('tsqr-8d16e396b237bf7a731333130d310cb9_QR_st1', 7, 0),
          task=(qr, (_apply_random, 'random_sample', 1310656009, (1000, 1000), (), {})),
          metric=1,
          cache_time=1454368444.49662,
          free_time=1454368446.769452)
```

These can be analyzed separately, or viewed in a bokeh plot using the provided `visualize` method on each profiler:

```
>>> prof.visualize()
```

To view multiple profilers at the same time, the `dask.diagnostics.visualize` function can be used. This takes a list of profilers, and creates a vertical stack of plots aligned along the x-axis:

```
>>> from dask.diagnostics import visualize
>>> visualize([prof, rprof, cprof])
```

Looking at the above figure, from top to bottom:

1. The results from the `Profiler` object. This shows the execution time for each task as a rectangle, organized along the y-axis by worker (in this case threads). Similar tasks are grouped by color, and by hovering over each task one can see the key and task that each block represents.
2. The results from the `ResourceProfiler` object. This shows two lines, one for total CPU percentage used by all the workers, and one for total memory usage.

3. The results from the `CacheProfiler` object. This shows a line for each task group, plotting the sum of the current `metric` in the cache against time. In this case it's the default metric (count), and the lines represent the number of each object in the cache at time. Note that the grouping and coloring is the same as for the `Profiler` plot, and that the task represented by each line can be found by hovering over the line.

From these plots we can see that the initial tasks (calls to `numpy.random.random` and `numpy.linalg.qr` for each chunk) are run concurrently, but only use slightly more than 100% CPU. This is because the call to `numpy.linalg.qr` currently doesn't release the global interpreter lock, so those calls can't truly be done in parallel. Next, there's a reduction step where all the blocks are combined. This requires all the results from the first step to be held in memory, as shown by the increased number of results in the cache, and increase in memory usage. Immediately after this task ends, the number of elements in the cache decreases, showing that they were only needed for this step. Finally, there's an interleaved set of calls to `dot` and `sum`. Looking at the CPU plot shows that these run both concurrently and in parallel, as the CPU percentage spikes up to around 250%.

4.16.4 Custom Callbacks

Custom diagnostics can be created using the callback mechanism described above. To add your own, subclass the `Callback` class, and define your own methods. Here we create a class that prints the name of every key as it's computed:

```
from dask.callbacks import Callback
class PrintKeys(Callback):
    def _pretask(self, key, dask, state):
        """Print the key of every task as it's started"""
        print("Computing: {0}!".format(repr(key)))
```

This can now be used as a context manager during computation:

```
>>> from operator import add, mul
>>> dsk = {'a': (add, 1, 2), 'b': (add, 3, 'a'), 'c': (mul, 'a', 'b')}
>>> with PrintKeys():
...     get(dsk, 'c')
Computing 'a'!
Computing 'b'!
Computing 'c'!
```

Alternatively, functions may be passed in as keyword arguments to `Callback`:

```
>>> def printkeys(key, dask, state):
...     print("Computing: {0}!".format(repr(key)))
>>> with Callback(pretask=printkeys):
...     get(dsk, 'c')
Computing 'a'!
Computing 'b'!
Computing 'c'!
```

Help & reference

- Support
- Development Guidelines
- Frequently Asked Questions
- Comparison to PySpark
- Opportunistic Caching
- Internal Data Ingestion

- Citations

4.17 Support

4.17.1 User questions and discussion

Discuss and ask questions on StackOverflow using the #dask tag.

4.17.2 Report a bug

Open issues with dask are tracked on GitHub at https://github.com/dask/dask/issues. If you find a bug, please:

1. Search issues to see if yours has already been reported. If no one has,
2. Create a new issue to report it!

4.17.3 Join the dask mailing list

We welcome you to join the dask mailing list, where users and the project developers:

- ask and answer questions
- discuss ways to use dask
- collaborate on solutions and approaches, and
- explore possible new features.

4.17.4 Chat with other users

Dask has a gitter chat room.

4.17.5 Get paid support

Dask is an open source project that was started by and is currently primarily developed by employees of Continuum Analytics, Inc.. In addition to the previous options, Continuum offers paid training and support:

- Python training — online and on-premises.
- Consulting — leverage our expertise to solve your analytics puzzles.
- Paid support plans — helping you get the most out of Python data tools.

4.17.6 Give us feedback

Help us make the documentation better! Please send feedback about the dask documentation to documentation@continuum.io.

4.18 Development Guidelines

Dask is a community maintained project. We welcome contributions in the form of bug reports, documentation, code, design proposals, and more. This page provides resources on how best to contribute.

4.18.1 Where to ask for help

Dask conversation happens in the following places:

1. StackOverflow #dask tag: for usage questions
2. Github Issue Tracker: for discussions around new features or established bugs
3. Gitter chat: for real-time discussion

For usage questions and bug reports we strongly prefer the use of StackOverflow and Github issues over gitter chat. Github and StackOverflow are more easily searchable by future users and so is more efficient for everyone's time. Gitter chat is generally reserved for community discussion.

4.18.2 Separate Code Repositories

Dask maintains code and documentation in a few git repositories hosted on the Github `dask` organization, http://github.com/dask. This includes the primary repository and several other repositories for different components. A non-exhaustive list follows:

- http://github.com/dask/dask: The main code repository holding parallel algorithms, the single-machine scheduler, and most documentation.
- http://github.com/dask/distributed: The distributed memory scheduler
- http://github.com/dask/hdfs3: Hadoop Filesystem interface
- http://github.com/dask/s3fs: S3 Filesystem interface
- http://github.com/dask/dask-ec2: AWS launching
- ...

Git and Github can be challenging at first. Fortunately good materials exist on the internet. Rather than repeat these materials here we refer you to Pandas' documentation and links on this subject at http://pandas.pydata.org/pandas-docs/stable/contributing.html

4.18.3 Issues

The community discusses and tracks known bugs and potential features in the Github Issue Tracker. If you have a new idea or have identified a bug then you should raise it there to start public discussion.

If you are looking for an introductory issue to get started with development then check out the introductory label, which contains issues that are good for starting developers. Generally familiarity with Python, NumPy, Pandas, and some parallel computing are assumed.

4.18.4 Development Environment

Download code

Clone the main dask git repostory (or whatever repository you're working on.):

```
git clone git@github.com:dask/dask.git
```

Install

You may want to install larger dependencies like NumPy and Pandas using a binary package manager, like conda. You can skip this step if you already have these libraries, don't care to use them, or have sufficient build environment on your computer to compile them when installing with `pip`:

```
conda install -y numpy pandas scipy bokeh cytoolz pytables h5py
```

Install dask and dependencies:

```
cd dask
pip install -e .[complete]
```

For development dask uses the following additional dependencies:

```
pip install pytest moto mock
```

Run Tests

Dask uses py.test for testing. You can run tests from the main dask directory as follows:

```
py.test dask --verbose
```

4.18.5 Contributing to Code

Dask maintains development standards that are similar to most PyData projects. These standards include language support, testing, documentation, and style.

Python Versions

Dask supports Python versions 2.7, 3.3, 3.4, and 3.5 in a single codebase. Name changes are handled by the `dask/compatibility.py` file.

Test

Dask employs extensive unit tests to ensure correctness of code both for today and for the future. Test coverage is expected for all code contributions.

Tests are written in a py.test style with bare functions.

```
def test_fibonacci():
    assert fib(0) == 0
    assert fib(1) == 0
    assert fib(10) == 55
    assert fib(8) == fib(7) + fib(6)

    for x in [-3, 'cat', 1.5]:
        with pytest.raises(ValueError):
            fib(x)
```

These tests should compromise well between covering all branches and fail cases and running quickly (slow test suites get run less often.)

You can run tests locally by running `py.test` in the local dask directory:

```
py.test dask --verbose
```

You can also test certain modules or individual tests for faster response:

```
py.test dask/dataframe --verbose

py.test dask/dataframe/tests/test_dataframe_core.py::test_set_index
```

Tests run automatically on the Travis.ci continuous testing framework on every push to every pull request on GitHub.

Docstrings

User facing functions should roughly follow the numpydoc standard, including sections for `Parameteters`, `Examples` and general explanatory prose.

By default examples will be doc-tested. Reproducible examples in documentation is valuable both for testing and, more importantly, for communication of common usage to the user. Documentation trumps testing in this case and clear examples should take precedence over using the docstring as testing space. To skip a test in the examples add the comment `# doctest: +SKIP` directly after the line.

```
def fib(i):
    """ A single line with a brief explanation

    A more thorough description of the function, consisting of multiple
    lines or paragraphs.

    Parameters
    ----------
    i: int
         A short description of the argument if not immediately clear

    Examples
    --------
    >>> fib(4)
    3
    >>> fib(5)
    5
    >>> fib(6)
    8
    >>> fib(-1)  # Robust to bad inputs
    ValueError(...)
    """
```

Docstrings are currently tested under Python 2.7 on travis.ci. You can test docstrings with pytest as follows:

```
py.test dask --doctest-modules
```

Style

Dask adheres loosely to PEP8 with rule-breaking allowed.

4.18.6 Contributing to Documentation

Dask uses Sphinx for documentation, hosted on http://readthedocs.org . Documentation is maintained in the RestructuredText markup language (`.rst` files) in `dask/docs/source`. The documentation consists both of prose and API documentation.

To build the documentation locally, first install requirements:

```
cd docs/
pip install -r requirements_docs.txt
```

Then build documentation with `make`:

```
make html
```

The resulting HTML files end up in the `build/html` directory.

You can now make edits to rst files and run `make html` again to update the affected pages.

4.19 Frequently Asked Questions

1. **Q: How do I debug my program when using dask?**

   If you want to inspect the dask graph itself see inspect docs.

   If you want to dive down with a Python debugger a common cause of frustration is the asynchronous schedulers which, because they run your code on different workers, are unable to provide access to the Python debugger. Fortunately you can change to a synchronous scheduler like `dask.get` or `dask.async.get_sync` by providing a `get=` keyword to the `compute` method:

   ```
   my_array.compute(get=dask.async.get_sync)
   ```

   Both `dask.async.get_sync` and `dask.get` will provide traceback traversals. `dask.async.get_sync` uses the same machinery of the async schedulers but with only one worker. `dask.get` is dead-simple but does not cache data and so can be slow for some workloads.

2. **Q: In ‘‘dask.array‘‘ what is ‘‘chunks‘‘?**

   Dask.array breaks your large array into lots of little pieces, each of which can fit in memory. `chunks` determines the size of those pieces.

   Users most often interact with chunks when they create an array as in:

   ```
   >>> x = da.from_array(dataset, chunks=(1000, 1000))
   ```

   In this case chunks is a tuple defining the shape of each chunk of your array; for example “Please break `dataset` into 1000 by 1000 chunks.”

   However internally dask uses a different representation, a tuple of tuples, to handle uneven chunk sizes that inevitably occur during computation.

3. **Q: How do I select a good value for ‘‘chunks‘‘?**

   Choosing good values for `chunks` can strongly impact performance. Here are some general guidelines. The strongest guide is memory:

   (a) The size of your blocks should fit in memory.

   (b) Actually, several blocks should fit in memory at once, assuming you want multi-core

(c) The size of the blocks should be large enough to hide scheduling overhead, which is a couple of milliseconds per task

(d) Generally I shoot for 10MB-100MB sized chunks

Additionally the computations you do may also inform your choice of `chunks`. Some operations like matrix multiply require anti-symmetric chunk shapes. Others like `svd` and `qr` only work on tall-and-skinny matrices with only a single chunk along all of the columns. Other operations might work but be faster or slower with different chunk shapes.

Note that you can `rechunk()` an array if necessary.

4. **Q: My computation fills memory, how do I spill to disk?**

   The schedulers endeavor not to use up all of your memory. However for some algorithms filling up memory is unavoidable. In these cases we can swap out the dictionary used to store intermediate results with a dictionary-like object that spills to disk. The Chest project handles this nicely.

```
>>> cache = Chest() # Uses temporary file. Deletes on garbage collection
```

   or

```
>>> cache = Chest(path='/path/to/dir', available_memory=8e9)  # Use 8GB
```

   This chest object works just like a normal dictionary but, when available memory runs out (defaults to 1GB) it starts pickling data and sending it to disk, retrieving it as necessary.

   You can specify your cache when calling `compute`

```
>>> x.dot(x.T).compute(cache=cache)
```

   Alternatively you can set your cache as a global option.

```
>>> with dask.set_options(cache=cache):  # sets state within with block
...     y = x.dot(x.T).compute()
```

   or

```
>>> dask.set_options(cache=cache)  # sets global state
>>> y = x.dot(x.T).compute()
```

   However, while using an on-disk cache is a great fallback performance, it's always best if we can keep from spilling to disk. You could try one of the following

   (a) Use a smaller chunk/partition size

   (b) If you are convinced that a smaller chunk size will not help in your case you could also report your problem on our issue tracker and work with the dask development team to improve our scheduling policies.

5. **How does Dask serialize functions?**

   When operating with the single threaded or multithreaded scheduler no function serialization is necessary. When operating with the distributed memory or multiprocessing scheduler Dask uses cloudpickle to serialize functions to send to worker processes. cloudpickle supports almost any kind of function, including lambdas, closures, partials and functions defined interactively.

   Cloudpickle can not serialize things like iterators, open files, locks, or other objects that are heavily tied to your current process. Attempts to serialize these objects (or functions that implicitly rely on these objects) will result in scheduler errors. You can verify that your objects are easily serializable by running them through the `cloudpickle.dumps/loads` functions

```
    from cloudpickle import dumps, loads
    obj2 = loads(dumps(obj))
    assert obj2 == obj
```

4.20 Comparison to PySpark

Spark is a popular distributed computing tool with a decent Python API PySpark. Spark is growing to become a dominant name today in Big Data analysis alongside Hadoop, for which MRJob is possibly the dominant Python layer.

Dask has several elements that appear to intersect this space and we are often asked, "How does Dask compare with Spark?"

Answering such comparison questions in an unbiased and informed way is hard, particularly when the differences can be somewhat technical. This document tries to do this; we welcome any corrections.

4.20.1 Summary

Apache Spark is an all-inclusive framework combining distributed computing, SQL queries, machine learning, and more that runs on the JVM and is commonly co-deployed with other Big Data frameworks like Hadoop. It was originally optimized for bulk data ingest and querying common in data engineering and business analytics but has since broadened out. Spark is typically used on small to medium sized cluster but also runs well on a single machine.

Dask is a parallel programming library that combines with the Numeric Python ecosystem to provide parallel arrays, dataframes, machine learning, and custom algorithms. It is based on Python and the foundational C/Fortran stack. Dask was originally designed to complement other libraries with parallelism, particular for numeric computing and advanced analytics, but has since broadened out. Dask is typically used on a single machine, but also runs well on a distributed cluster.

Generally Dask is smaller and lighter weight than Spark. This means that it has fewer features and instead is intended to be used in conjunction with other libraries, particularly those in the numeric Python ecosystem.

4.20.2 User-Facing Differences

Scale

Spark began its life aimed at the thousand node cluster case. As such it thinks well about worker failures and integration with data-local file systems like the Hadoop FileSystem (HDFS). That being said, Spark can run in standalone mode on a single machine.

Dask began its life building out parallel algorithms for numerical array computations on a single computer. As such it thinks well about low-latency scheduling, low memory footprints, shared memory, and efficient use of local disk. That being said dask can run on a distributed cluster, making use of HDFS and other Big Data technologies.

Java Python Performance

Spark is written in Scala, a multi-paradigm language built on top of the Java Virtual Machine (JVM). Since the rise of Hadoop, Java based languages have steadily gained traction on data warehousing tasks and are good at managing large amounts of heterogeneous data such as you might find in JSON blobs. The Spark development team is now focusing more on binary and native data formats with their new effort, Tungsten.

Dask is written in Python, a multi-paradigm language built on top of the C/Fortran native stack. This stack benefits from decades of scientific research optimizing very fast computation on numeric data. As such, dask is already very good on analytic computations on data such as you might find in HDF5 files or analytic databases. It can also handle JSON blob type data using Python data structures (which are surprisingly fast) using the cytoolz library in parallel.

Java Python Disconnect

Python users on Spark sometimes express frustration by how far separated they are from computations. Some of this is inevitable; distributed debugging is a hard problem. Some of it however is due to having to hop over the JVM. Spark workers spin up JVMs which in turn spin up Python processes. Data moving back and forth makes extra trips both through a distributed cluster and also through extra serialization layers (see py4j) and computation layers. Limitations like the Java heap size and large Java stack traces come as a surprise to users accustomed to native code execution.

Dask has an advantage for Python users because it is itself a Python library, so serialization and debugging when things go wrong happens more smoothly.

However, Dask only benefits Python users while Spark is useful in a variety of JVM languages (Scala, Java, Clojure) and also has limited support in Python and R. New Spark projects like the DataFrame skip serialization and boxed execution issues by forgoing the Python process entirely and instead have Python code drive native Scala code. APIs for these libraries tend to lag a bit behind their Scala counterparts.

Scope

Spark was originally built around the RDD, an unordered collection allowing repeats. Most spark add-ons were built on top of this construct, inheriting both its abilities and limitations.

Dask is built on a lower-level and more general construct of a generic task graph with arbitrary data dependencies. This allows more general computations to be built by users within the dask framework. This is probably the largest fundamental difference between the two projects. Dask gives up high-level understanding to allow users to express more complex parallel algorithms. This ended up being essential when writing complex projects like `dask.array`, datetime algorithms in `dask.dataframe` or non-trivial algorithms in machine learning.handling in .

4.20.3 Developer-Facing Differences

Graph Granularity

Both Spark and Dask represent computations with directed acyclic graphs. These graphs however represent computations at very different granularities.

One operation on a Spark RDD might add a node like `Map` and `Filter` to the graph. These are high-level operations that convey meaning and will eventually be turned into many little tasks to execute on individual workers. This many-little-tasks state is only available internally to the Spark scheduler.

Dask graphs skip this high-level representation and go directly to the many-little-tasks stage. As such one `map` operation on a dask collection will immediately generate and add possibly thousands of tiny tasks to the dask graph.

This difference in the scale of the underlying graph has implications on the kinds of analysis and optimizations one can do and also on the generality that one exposes to users. Dask is unable to perform some optimizations that Spark can because Dask schedulers do not have a top-down picture of the computation they were asked to perform. However, dask is able to easily represent far more complex algorithms and expose the creation of these algorithms to normal users.

Dask.bag, the equivalent of the Spark.RDD, is just one abstraction built on top of dask. Others exist. Alternatively power-users can forego high-level collections entirely and jump straight to direct low-level task scheduling.

Coding Styles

Both Spark and Dask are written in a functional style. Spark will probably be more familiar to those who enjoy algebraic types while dask will probably be more familiar to those who enjoy Lisp and "code as data structures".

4.20.4 Conclusion

Spark is mature and all-inclusive. If you want a single project that does everything and you're already on Big Data hardware then Spark is a safe bet, especially if your use cases are typical ETL + SQL and you're already using Scala.

Dask is lighter weight and is easier to integrate into existing code and hardware. If your problems vary beyond typical ETL + SQL and you want to add flexible parallelism to existing solutions then dask may be a good fit, especially if you are already using Python and associated libraries like NumPy and Pandas.

If you are looking to manage a terabyte or less of tabular CSV or JSON data then you should forget both Spark and Dask and use Postgres or MongoDB.

4.21 Opportunistic Caching

EXPERIMENTAL FEATURE added to Version 0.6.2 and above - see *disclaimer*.

Dask usually removes intermediate values as quickly as possible in order to make space for more data to flow through your computation. However, in some cases, we may want to hold onto intermediate values, because they might be useful for future computations in an interactive session.

We need to balance the following concerns:

1. Intermediate results might be useful in future unknown computations
2. Intermediate results also fill up memory, reducing space for the rest of our current computation.

Negotiating between these two concerns helps us to leverage the memory that we have available to speed up future, unanticipated computations. Which intermediate results should we keep?

This document explains an experimental, opportunistic caching mechanism that automatically picks out and stores useful tasks.

4.21.1 Motivating Example

Consider computing the maximum value of a column in a CSV file:

```
>>> import dask.dataframe as dd
>>> df = dd.read_csv('myfile.csv')
>>> df.columns
['first-name', 'last-name', 'amount', 'id', 'timestamp']

>>> df.amount.max().compute()
1000
```

Even though our full dataset may be too large to fit in memory, the single `df.amount` column may be small enough to hold in memory just in case it might be useful in the future. This is often the case during data exploration, because we investigate the same subset of our data repeatedly before moving on.

For example, we may now want to find the minimum of the amount column:

```
>>> df.amount.min().compute()
-1000
```

Under normal operations, this would need to read through the entire CSV file over again. This is somewhat wasteful, and stymies interactive data exploration.

4.21.2 Two Simple Solutions

If we know ahead of time that we want both the maximum and minimum, we can compute them simultaneously. Dask will share intermediates intelligently, reading through the dataset only once:

```
>>> dd.compute(df.amount.max(), df.amount.min())
(1000, -1000)
```

If we know that this column fits in memory then we can also explicitly compute the column and then continue forward with straight Pandas:

```
>>> amount = df.amount.compute()
>>> amount.max()
1000
>>> amount.min()
-1000
```

If either of these solutions work for you, great. Otherwise, continue on for a third approach.

4.21.3 Automatic Opportunistic Caching

Another approach is to watch *all* intermediate computations, and *guess* which ones might be valuable to keep for the future. Dask has an *opportunistic caching mechanism* that stores intermediate tasks that show the following characteristics:

1. Expensive to compute
2. Cheap to store
3. Frequently used

We can activate a fixed sized cache as a callback.

```
>>> from dask.cache import Cache
>>> cache = Cache(2e9)  # Leverage two gigabytes of memory
>>> cache.register()    # Turn cache on globally
```

Now the cache will watch every small part of the computation and judge the value of that part based on the three characteristics listed above (expensive to compute, cheap to store, and frequently used).

Dask will hold on to 2GB of the best intermediate results it can find, evicting older results as better results come in. If the `df.amount` column fits in 2GB then probably all of it will be stored while we keep working on it.

If we start work on something else, then the `df.amount` column will likely be evicted to make space for other more timely results:

```
>>> df.amount.max().compute()  # slow the first time
1000
>>> df.amount.min().compute()  # fast because df.amount is in the cache
-1000
>>> df.id.nunique().compute()  # starts to push out df.amount from cache
```

4.21.4 Cache tasks, not expressions

This caching happens at the low-level scheduling layer, not the high-level dask.dataframe or dask.array layer. We don't explicitly cache the column `df.amount`. Instead, we cache the hundreds of small pieces of that column that form the dask graph. It could be that we end up caching only a fraction of the column.

This means that the opportunistic caching mechanism described above works for *all* dask computations, as long as those computations employ a consistent naming scheme (as all of dask.dataframe, dask.array, and dask.delayed do.)

You can see which tasks are held by the cache by inspecting the following attributes of the cache object:

```
>>> cache.cache.data
<stored values>
>>> cache.cache.heap.heap
<scores of items in cache>
>>> cache.cache.nbytes
<number of bytes per item in cache>
```

The cache object is powered by cachey, a tiny library for opportunistic caching.

4.21.5 Disclaimer

This feature is still experimental, and can cause your computation to fill up RAM.

Restricting your cache to a fixed size like 2GB requires dask to accurately count the size of each of our objects in memory. This can be tricky, particularly for Pythonic objects like lists and tuples, and for DataFrames that contain object dtypes.

It is entirely possible that the caching mechanism will *undercount* the size of objects, causing it to use up more memory than anticipated which can lead to blowing up RAM and crashing your session.

4.22 Internal Data Ingestion

Dask contains internal tools for extensible data ingestion in the `dask.bytes` package. *These functions are developer-focused rather than for direct consumption by users. These functions power user facing functions like ''dd.read_csv'' and ''db.read_text'' which are probably more useful for most users.*

read_bytes(urlpath[, delimiter, not_zero, ...])	Convert path to a list of delayed values
open_files(urlpath[, compression])	Given path return dask.delayed file-like objects
open_text_files(urlpath[, encoding, errors, ...])	Given path return dask.delayed file-like objects in text mode

These functions are extensible in their output formats (bytes, file objects), their input locations (file system, S3, HDFS), line delimiters, and compression formats.

These functions provide data as `dask.delayed` objects. These objects either point to blocks of bytes (`read_bytes`) or open file objects (`open_files`, `open_text_files`). They can handle different compression formats by prepending protocols like `s3://` or `hdfs://`. They handle compression formats listed in the `dask.bytes.compression` module.

These functions are not used for all data sources. Some data sources like HDF5 are quite particular and receive custom treatment.

4.22.1 Delimiters

The `read_bytes` function takes a path (or globstring of paths) and produces a sample of the first file and a list of delayed objects for each of the other files. If passed a delimiter such as `delimiter=b'\n'` it will ensure that the blocks of bytes start directly after a delimiter and end directly before a delimiter. This allows other functions, like `pd.read_csv`, to operate on these delayed values with expected behavior.

These delimiters are useful both for typical line-based formats (log files, CSV, JSON) as well as other delimited formats like Avro, which may separate logical chunks by a complex sentinel string.

4.22.2 Locations

These functions dispatch to other functions that handle different storage backends, like S3 and HDFS. These storage backends register themselves with protocols and so are called whenever the path is prepended with a string like the following:

```
s3://bucket/keys-*.csv
```

4.22.3 Compression

These functions support widely available compression technologies like `gzip`, `bz2`, `xz`, `snappy`, and `lz4`. More compressions can be easily added by inserting functions into dictionaries available in the `dask.bytes.compression` module. This can be done at runtime and need not be added directly to the codebase.

However, not all compression technologies are available for all functions. In particular, compression technologies like `gzip` do not support efficient random access and so are useful for streaming `open_files` but not useful for `read_bytes` which splits files at various points.

4.22.4 Functions

`dask.bytes.`**`read_bytes`**(*urlpath*, *delimiter=None*, *not_zero=False*, *blocksize=134217728*, *sample=True*, *compression=None*, ***kwargs*)

Convert path to a list of delayed values

The path may be a filename like `'2015-01-01.csv'` or a globstring like `'2015-*-*.csv'`.

The path may be preceded by a protocol, like `s3://` or `hdfs://` if those libraries are installed.

This cleanly breaks data by a delimiter if given, so that block boundaries start directly after a delimiter and end on the delimiter.

Parameters **urlpath: string**

Absolute or relative filepath, URL (may include protocols like `s3://`), or globstring pointing to data.

delimiter: bytes An optional delimiter, like ‘‘b’

‘‘‘ **on which to split blocks of bytes**

not_zero: force seek of start-of-file delimiter, discarding header blocksize: int (=128MB)

Chunk size

compression: string or None String like 'gzip' or 'xz'. Must support efficient random access.

sample: bool, int Whether or not to return a sample from the first 10k bytes

****kwargs: dict** Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns 10kB sample header and list of `dask.Delayed` objects or list of lists of delayed objects if `fn` is a globstring.

Examples

```
>>> sample, blocks = read_bytes('2015-*-*.csv', delimiter=b'\n')
>>> sample, blocks = read_bytes('s3://bucket/2015-*-*.csv', delimiter=b'\n')
```

`dask.bytes.`**`open_files`**(*urlpath*, *compression=None*, ***kwargs*)

Given path return dask.delayed file-like objects

Parameters **urlpath: string**

Absolute or relative filepath, URL (may include protocols like `s3://`), or globstring pointing to data.

compression: string

Compression to use. See `dask.bytes.compression.files` for options.

****kwargs: dict**

Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns List of `dask.delayed` objects that compute to file-like objects

Examples

```
>>> files = open_files('2015-*-*.csv')
>>> files = open_files('s3://bucket/2015-*-*.csv.gz', compression='gzip')
```

`dask.bytes.`**`open_text_files`**(*urlpath*, *encoding='utf-8'*, *errors='strict'*, *compression=None*, ***kwargs*)

Given path return dask.delayed file-like objects in text mode

Parameters **urlpath: string**

Absolute or relative filepath, URL (may include protocols like `s3://`), or globstring pointing to data.

encoding: string

errors: string

compression: string

Compression to use. See `dask.bytes.compression.files` for options.

****kwargs: dict**

Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns List of `dask.delayed` objects that compute to text file-like objects

Examples

```
>>> files = open_text_files('2015-*-*.csv', encoding='utf-8')
>>> files = open_text_files('s3://bucket/2015-*-*.csv')
```

4.23 Citations

Dask is developed by many people from many institutions. Some of these developers are academics who depend on academic citations to justify their efforts. Unfortunately, no single citation can do all of these developers (and the developers to come) sufficient justice. Instead, we choose to use a single blanket citation for all developers past and present.

To cite Dask in publications, please use the following:

```
Dask Development Team (2016). Dask: Library for dynamic task scheduling
URL http://dask.pydata.org
```

A BibTeX entry for LaTeX users follows:

```
@Manual{,
  title = {Dask: Library for dynamic task scheduling},
  author = {{Dask Development Team}},
  year = {2016},
  url = {http://dask.pydata.org},
}
```

The full author list is available using git, or by looking at the AUTHORS file.

4.23.1 Papers about parts of Dask

Rocklin, Matthew. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling." (2015). PDF.

```
@InProceedings{ matthew_rocklin-proc-scipy-2015,
  author    = { Matthew Rocklin },
  title     = { Dask: Parallel Computation with Blocked algorithms and Task Scheduling },
  booktitle = { Proceedings of the 14th Python in Science Conference },
  pages     = { 130 - 136 },
  year      = { 2015 },
  editor    = { Kathryn Huff and James Bergstra }
}
```

Contact

- For user questions please tag StackOverflow questions with the #dask tag.
- For bug reports and feature requests please use the GitHub issue tracker
- For community discussion please use blaze-dev@continuum.io
- For chat, see gitter chat room

Dask is part of the Blaze project supported and offered by Continuum Analytics and contributors under a 3-clause BSD license.

[R76] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R77] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arccosh

[R78] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R79] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arcsinh

[R80] ISO/IEC standard 9899:1999, “Programming language C.”

[R81] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R82] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arctanh

[R83] “Lecture Notes on the Status of IEEE 754”, William Kahan, http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

[R84] “How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?”, William Kahan, http://www.cs.berkeley.edu/~wkahan/Mindless.pdf

[R85] Wikipedia, “Exponential function”, http://en.wikipedia.org/wiki/Exponential_function

[R86] M. Abramovitz and I. A. Stegun, “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables,” Dover, 1964, p. 69, http://www.math.sfu.ca/~cbm/aands/page_69.htm

[R87] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R88] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[R89] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R90] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[R91] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R92] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[R93] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. http://www.math.sfu.ca/~cbm/aands/

[R94] Wikipedia, “Hyperbolic function”, http://en.wikipedia.org/wiki/Hyperbolic_function

[CT] Cooley, James W., and John W. Tukey, 1965, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19: 297-301.

[R95] Dalgaard, Peter, "Introductory Statistics with R", Springer-Verlag, 2002.

[R96] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill, Fifth Edition, 2002.

[R97] Lentner, Marvin, "Elementary Applied Statistics", Bogden and Quigley, 1972.

[R98] Weisstein, Eric W. "Binomial Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialDistribution.html

[R99] Wikipedia, "Binomial-distribution", http://en.wikipedia.org/wiki/Binomial_distribution

[R100] Peyton Z. Peebles Jr., "Probability, Random Variables and Random Signal Principles", 4th ed, 2001, p. 57.

[R101] "Poisson Process", Wikipedia, http://en.wikipedia.org/wiki/Poisson_process

[R102] "Exponential Distribution, Wikipedia, http://en.wikipedia.org/wiki/Exponential_distribution

[R103] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill, Fifth Edition, 2002.

[R104] Wikipedia, "F-distribution", http://en.wikipedia.org/wiki/F-distribution

[R105] Weisstein, Eric W. "Gamma Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[R106] Wikipedia, "Gamma-distribution", http://en.wikipedia.org/wiki/Gamma-distribution

[R107] Lentner, Marvin, "Elementary Applied Statistics", Bogden and Quigley, 1972.

[R108] Weisstein, Eric W. "Hypergeometric Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HypergeometricDistribution.html

[R109] Wikipedia, "Hypergeometric-distribution", http://en.wikipedia.org/wiki/Hypergeometric-distribution

[R110] Abramowitz, M. and Stegun, I. A. (Eds.). Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing. New York: Dover, 1972.

[R111] The Laplace distribution and generalizations By Samuel Kotz, Tomasz J. Kozubowski, Krzysztof Podgorski, Birkhauser, 2001.

[R112] Weisstein, Eric W. "Laplace Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LaplaceDistribution.html

[R113] Wikipedia, "Laplace distribution", http://en.wikipedia.org/wiki/Laplace_distribution

[R114] Reiss, R.-D. and Thomas M. (2001), Statistical Analysis of Extreme Values, from Insurance, Finance, Hydrology and Other Fields, Birkhauser Verlag, Basel, pp 132-133.

[R115] Weisstein, Eric W. "Logistic Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LogisticDistribution.html

[R116] Wikipedia, "Logistic-distribution", http://en.wikipedia.org/wiki/Logistic-distribution

[R117] Buzas, Martin A.; Culver, Stephen J., Understanding regional species diversity through the log series distribution of occurrences: BIODIVERSITY RESEARCH Diversity & Distributions, Volume 5, Number 5, September 1999 , pp. 187-195(9).

[R118] Fisher, R.A,, A.S. Corbet, and C.B. Williams. 1943. The relation between the number of species and the number of individuals in a random sample of an animal population. Journal of Animal Ecology, 12:42-58.

[R119] D. J. Hand, F. Daly, D. Lunn, E. Ostrowski, A Handbook of Small Data Sets, CRC Press, 1994.

[R120] Wikipedia, "Logarithmic-distribution", http://en.wikipedia.org/wiki/Logarithmic-distribution

[R121] Weisstein, Eric W. "Negative Binomial Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NegativeBinomialDistribution.html

[R122] Wikipedia, "Negative binomial distribution", http://en.wikipedia.org/wiki/Negative_binomial_distribution

[R123] Delhi, M.S. Holla, "On a noncentral chi-square distribution in the analysis of weapon systems effectiveness", Metrika, Volume 15, Number 1 / December, 1970.

[R124] Wikipedia, "Noncentral chi-square distribution" http://en.wikipedia.org/wiki/Noncentral_chi-square_distribution

[R125] Wikipedia, "Normal distribution", http://en.wikipedia.org/wiki/Normal_distribution

[R126] P. R. Peebles Jr., "Central Limit Theorem" in "Probability, Random Variables and Random Signal Principles", 4th ed., 2001, pp. 51, 51, 125.

[R127] Francis Hunt and Paul Johnson, On the Pareto Distribution of Sourceforge projects.

[R128] Pareto, V. (1896). Course of Political Economy. Lausanne.

[R129] Reiss, R.D., Thomas, M.(2001), Statistical Analysis of Extreme Values, Birkhauser Verlag, Basel, pp 23-30.

[R130] Wikipedia, "Pareto distribution", http://en.wikipedia.org/wiki/Pareto_distribution

[R131] Weisstein, Eric W. "Poisson Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PoissonDistribution.html

[R132] Wikipedia, "Poisson distribution", http://en.wikipedia.org/wiki/Poisson_distribution

[R133] Christian Kleiber, Samuel Kotz, "Statistical size distributions in economics and actuarial sciences", Wiley, 2003.

[R134] Heckert, N. A. and Filliben, James J. (2003). NIST Handbook 148: Dataplot Reference Manual, Volume 2: Let Subcommands and Library Functions", National Institute of Standards and Technology Handbook Series, June 2003. http://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/powpdf.pdf

[R135] Brighton Webs Ltd., Rayleigh Distribution, http://www.brighton-webs.co.uk/distributions/rayleigh.asp

[R136] Wikipedia, "Rayleigh distribution" http://en.wikipedia.org/wiki/Rayleigh_distribution

[R137] NIST/SEMATECH e-Handbook of Statistical Methods, "Cauchy Distribution", http://www.itl.nist.gov/div898/handbook/eda/section3/eda3663.htm

[R138] Weisstein, Eric W. "Cauchy Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CauchyDistribution.html

[R139] Wikipedia, "Cauchy distribution" http://en.wikipedia.org/wiki/Cauchy_distribution

[R140] Weisstein, Eric W. "Gamma Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[R141] Wikipedia, "Gamma-distribution", http://en.wikipedia.org/wiki/Gamma-distribution

[R142] Dalgaard, Peter, "Introductory Statistics With R", Springer, 2002.

[R143] Wikipedia, "Student's t-distribution" http://en.wikipedia.org/wiki/Student's_t-distribution

[R144] Wikipedia, "Triangular distribution" http://en.wikipedia.org/wiki/Triangular_distribution

[R145] Brighton Webs Ltd., Wald Distribution, http://www.brighton-webs.co.uk/distributions/wald.asp

[R146] Chhikara, Raj S., and Folks, J. Leroy, "The Inverse Gaussian Distribution: Theory : Methodology, and Applications", CRC Press, 1988.

[R147] Wikipedia, "Wald distribution" http://en.wikipedia.org/wiki/Wald_distribution

[R148] Waloddi Weibull, Professor, Royal Technical University, Stockholm, 1939 “A Statistical Theory Of The Strength Of Materials”, Ingeniorsvetenskapsakademiens Handlingar Nr 151, 1939, Generalstabens Litografiska Anstalts Forlag, Stockholm.

[R149] Waloddi Weibull, 1951 “A Statistical Distribution Function of Wide Applicability”, Journal Of Applied Mechanics ASME Paper.

[R150] Wikipedia, “Weibull distribution”, http://en.wikipedia.org/wiki/Weibull_distribution

[R151] Pebay, Philippe (2008), “Formulas for Robust, One-Pass Parallel

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z