

---

# **dask Documentation**

*Release 0.16.1*

**Dask Development Team**

**Jan 25, 2018**



---

## Getting Started

---

<b>1</b>	<b>Familiar user interface</b>	<b>3</b>
<b>2</b>	<b>Scales from laptops to clusters</b>	<b>5</b>
<b>3</b>	<b>Complex Algorithms</b>	<b>7</b>
<b>4</b>	<b>Index</b>	<b>9</b>
	<b>Bibliography</b>	<b>617</b>



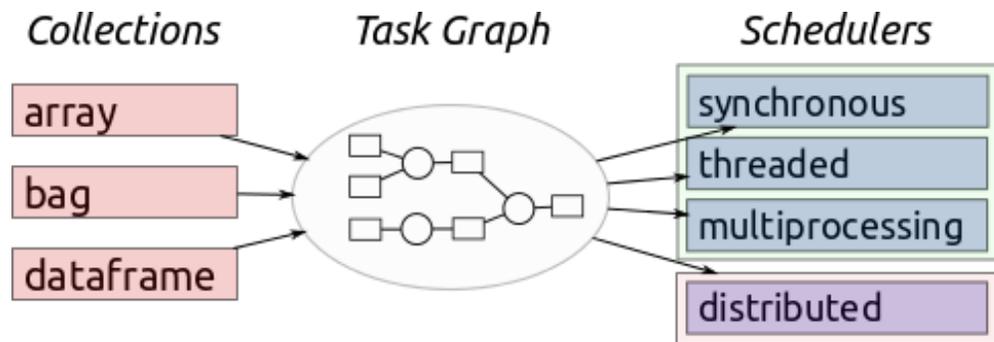
*Dask is a flexible parallel computing library for analytic computing.*

Dask is composed of two components:

1. **Dynamic task scheduling** optimized for computation. This is similar to *Airflow*, *Luigi*, *Celery*, or *Make*, but optimized for interactive computational workloads.
2. **“Big Data” collections** like parallel arrays, dataframes, and lists that extend common interfaces like *NumPy*, *Pandas*, or *Python iterators* to larger-than-memory or distributed environments. These parallel collections run on top of the dynamic task schedulers.

Dask emphasizes the following virtues:

- **Familiar:** Provides parallelized NumPy array and Pandas DataFrame objects
- **Flexible:** Provides a task scheduling interface for more custom workloads and integration with other projects.
- **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
- **Scales up:** Runs resiliently on clusters with 1000s of cores
- **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



See the [dask.distributed documentation \(separate website\)](#) for more technical information on Dask’s distributed scheduler,



---

Familiar user interface

---

Dask DataFrame mimics Pandas - *documentation*

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

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

Dask Array mimics NumPy - *documentation*

```
import numpy as np
f = h5py.File('myfile.hdf5')
x = np.array(f['/small-data'])
x - x.mean(axis=1)

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

Dask Bag mimics iterators, Toolz, and PySpark - *documentation*

```
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 - *documentation*

```
from dask import delayed
L = []
for fn in filenames:
    data = delayed(load)(fn)
    L.append(delayed(process)(data))
result = delayed(summarize)(L)
result.compute()
```

The `concurrent.futures` interface provides general submission of custom tasks: - *documentation*

```
from dask.distributed import Client
client = Client('scheduler:port')
```

```
futures = []
for fn in filenames:
    future = client.submit(load, fn)
    futures.append(future)

summary = client.submit(summarize, futures)
summary.result()
```

---

### 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.



---

## 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.



### Getting Started

- *Install Dask*
- *Setup*
- *Use Cases*
- *Examples*
- *Community*

## 4.1 Install Dask

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

### 4.1.1 Anaconda

### 4.1.2 Conda

Dask is installed by default in [Anaconda](#):

You can update Dask using the `conda` command:

```
conda install dask
```

This installs Dask and **all** common dependencies, including Pandas and NumPy.

Dask packages are maintained both on the default channel and on [conda-forge](#).

Optionally, you can obtain a minimal dask installation using the following command:

```
conda install dask-core
```

This will install a minimal set of dependencies required to run dask, similar to (but not exactly the same as) `pip install dask` below.

### 4.1.3 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..)

### 4.1.4 Install from Source

To install dask from source, clone the repository from [github](https://github.com/dask/dask):

```
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`.

### 4.1.5 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 Setup

This page describes various ways to set up Dask on different hardware, either locally on your own machine or on a distributed cluster. If you are just getting started then this page is unnecessary. Dask does not require any setup if you only want to use it on a single computer.

Dask has two families of task schedulers:

1. **Single machine scheduler:** This scheduler provides basic features on a local process or thread pool. This scheduler was made first and is the default. It is simple and cheap to use. It can only be used on a single machine and does not scale.
2. **Distributed scheduler:** This scheduler is more sophisticated, offers more features, but also requires a bit more effort to set up. It can run locally or distributed across a cluster.

If you import Dask, set up a computation, and then call `compute` then you will use the single-machine scheduler by default. To use the `dask.distributed` scheduler you must set up a `Client`

```
import dask.dataframe as dd
df = dd.read_csv(...)
df.x.sum().compute() # This uses the single-machine scheduler by default
```

```
from dask.distributed import Client
client = Client(...) # Connect to distributed cluster and override default
df.x.sum().compute() # This now runs on the distributed system
```

Note that the newer `dask.distributed` scheduler is often preferable even on single workstations. It contains many diagnostics and features not found in the older single-machine scheduler. The following pages explain in more detail how to set up Dask on a variety of local and distributed hardware.

- **Single Machine:**

- *Default Scheduler:* The no-setup default. Uses local threads or processes for larger-than-memory processing
- *Dask.distributed:* The sophistication of the newer system on a single machine. This provides more advanced features while still requiring almost no setup.

- **Distributed computing:**

- *Manual Setup:* The command line interface to set up `dask-scheduler` and `dask-worker` processes. Useful for IT or anyone building a deployment solution.
- *SSH:* Use SSH to set up Dask across an un-managed cluster
- *High Performance Computers:* How to run Dask on traditional HPC environments using tools like MPI, or job schedulers like SLURM, SGE, TORQUE, LSF, and so on
- *Kubernetes:* Deploy Dask on the popular Kubernetes resource manager. This is particularly useful for any cloud deployments on Google, Amazon, or Microsoft Azure.
- *Python API (advanced):* Create `Scheduler` and `Worker` objects from Python as part of a distributed Tornado TCP application. This page is useful for those building custom frameworks.

## 4.2.1 Single-Machine Scheduler

The default Dask scheduler provides parallelism on a single machine by using either threads or processes. It is the default choice used by Dask because it requires no setup. You don't need to make any choices or set anything up to use this scheduler, however you do have a choice between threads and processes:

1. **Threads:** Use multiple threads in the same process. This option is good for numeric code that releases the `GIL` (like NumPy, Pandas, Scikit-Learn, Numba, ...) because data is free to share. This is the default scheduler for `dask.array`, `dask.dataframe`, and `dask.delayed`
2. **Processes:** Send data to separate processes for processing. This option is good when operating on pure Python objects like strings or JSON-like dictionary data that holds onto the `GIL` but not very good when operating on

numeric data like Pandas dataframes or NumPy arrays. Using processes avoids GIL issues but can also result in a lot of inter-process communication, which can be slow. This is the default scheduler for `dask.bag` and is sometimes useful with `dask.dataframe`.

Note that the `dask.distributed` scheduler is often a better choice when working with GIL-bound code. See [Dask.distributed on a single machine](#).

3. **Single-threaded:** Execute computations in a single thread. This option provides no parallelism, but is useful when debugging or profiling. Turning your parallel execution into a sequential one can be a convenient option in many situations where you want to better understand what is going on.

## Selecting Threads, Processes, or Single Threaded

Currently these options are available by selecting different `get` functions:

- `dask.threaded.get`: The threaded scheduler
- `dask.multiprocessing.get`: The multiprocessing scheduler
- `dask.local.get_sync`: The single-threaded scheduler

You can specify these functions in any of the following ways:

- When calling `.compute()`

```
x.compute(get=dask.threaded.get)
```

- With a context manager

```
with dask.set_options(get=dask.threaded.get):
    x.compute()
    y.compute()
```

- As a global setting

```
dask.set_options(get=dask.threaded.get)
```

## Use the Distributed Scheduler

The newer `dask.distributed` scheduler also works well on a single machine and offers more features and diagnostics. See [this page](#) for more information.

### 4.2.2 Single Machine: Dask.distributed

The `dask.distributed` scheduler works well on a single machine. It is sometimes preferred over the default scheduler for the following reasons:

1. It provides access to asynchronous API, notably *Futures*
2. It provides a diagnostic dashboard that can provide valuable insight on performance and progress
3. It handles data locality with more sophistication, and so can be more efficient than the multiprocessing scheduler on workloads that require multiple processes.

You can create a `dask.distributed` scheduler by importing and creating a `Client` with no arguments. This overrides whatever default was previously set.

```
from dask.distributed import Client
client = Client()
```

You can navigate to <http://localhost:8787/status> to see the diagnostic dashboard if you have Bokeh installed.

## Client

You can trivially set up a local cluster on your machine by instantiating a Dask Client with no arguments

```
from dask.distributed import Client
client = Client()
```

This sets up a scheduler in your local process and several processes running single-threaded Workers.

If you want to run workers in your same process you can pass the `processes=False` keyword argument.

```
client = Client(processes=False)
```

This is sometimes preferable if you want to avoid inter-worker communication and your computations release the GIL. This is common when primarily using NumPy or Dask.array.

## LocalCluster

The `Client()` call described above is shorthand for creating a `LocalCluster` and then passing that to your client.

```
from dask.distributed import Client, Cluster
cluster = LocalCluster()
client = Client(cluster)
```

This is equivalent, but somewhat more explicit. You may want to look at the keyword arguments available on `LocalCluster` to understand the options available to you on handling the mixture of threads and processes, specifying explicit ports, and so on.

```
class distributed.deploy.local.LocalCluster(n_workers=None,
                                             threads_per_worker=None, processes=True,
                                             loop=None, start=True, ip=None,
                                             scheduler_port=0, silence_logs=50,
                                             diagnostics_port=8787, services={},
                                             worker_services={}, **worker_kwargs)
```

Create local Scheduler and Workers

This creates a “cluster” of a scheduler and workers running on the local machine.

### Parameters `n_workers`: int

Number of workers to start

### `processes`: bool

Whether to use processes (True) or threads (False). Defaults to True

### `threads_per_worker`: int

Number of threads per each worker

### `scheduler_port`: int

Port of the scheduler. 8786 by default, use 0 to choose a random port

**silence\_logs: logging level**

Level of logs to print out to stdout. `logging.CRITICAL` by default. Use a falsey value like `False` or `None` for no change.

**ip: string**

IP address on which the scheduler will listen, defaults to only localhost

**kwargs: dict**

Extra worker arguments, will be passed to the `Worker` constructor.

**Examples**

```
>>> c = LocalCluster() # Create a local cluster with as many workers as cores
>>> c
LocalCluster("127.0.0.1:8786", workers=8, ncores=8)
```

```
>>> c = Client(c) # connect to local cluster
```

Add a new worker to the cluster `>>> w = c.start_worker(ncores=2) # doctest: +SKIP`

Shut down the extra worker `>>> c.remove_worker(w) # doctest: +SKIP`

**close** (*timeout=20*)

Close the cluster

**scale\_down** (*workers*)

Remove `workers` from the cluster

Given a list of worker addresses this function should remove those workers from the cluster. This may require tracking which jobs are associated to which worker address.

This can be implemented either as a function or as a Tornado coroutine.

**scale\_up** (*n, \*\*kwargs*)

Bring the total count of workers up to `n`

This function/coroutine should bring the total number of workers up to the number `n`.

This can be implemented either as a function or as a Tornado coroutine.

**start\_worker** (*ncores=0, \*\*kwargs*)

Add a new worker to the running cluster

**Parameters** **port: int (optional)**

Port on which to serve the worker, defaults to 0 or random

**ncores: int (optional)**

Number of threads to use. Defaults to number of logical cores

**Returns** The created `Worker` or `Nanny` object. Can be discarded.

**Examples**

```
>>> c = LocalCluster()
>>> c.start_worker(ncores=2)
```

**stop\_worker**(*w*)  
Stop a running worker

### Examples

```
>>> c = LocalCluster()
>>> w = c.start_worker(ncores=2)
>>> c.stop_worker(w)
```

## 4.2.3 Command Line

This is the most fundamental way to deploy Dask on multiple machines. In production environments this process is often automated by some other resource manager and so it is rare that people need to follow these instructions explicitly. Instead, these instructions are useful for IT professionals who may want to set up automated services to deploy Dask within their institution.

A `dask.distributed` network consists of one `dask-scheduler` process and several `dask-worker` processes that connect to that scheduler. These are normal Python processes that can be executed from the command line. We launch the `dask-scheduler` executable in one process and the `dask-worker` executable in several processes, possibly on different machines.

Launch `dask-scheduler` on one node:

```
$ dask-scheduler
Scheduler at:  tcp://192.0.0.100:8786
```

Then launch `dask-worker` on the rest of the nodes, providing the address to the node that hosts `dask-scheduler`:

```
$ dask-worker tcp://192.0.0.100:8786
Start worker at:  tcp://192.0.0.1:12345
Registered to:   tcp://192.0.0.100:8786

$ dask-worker tcp://192.0.0.100:8786
Start worker at:  tcp://192.0.0.2:40483
Registered to:   tcp://192.0.0.100:8786

$ dask-worker tcp://192.0.0.100:8786
Start worker at:  tcp://192.0.0.3:27372
Registered to:   tcp://192.0.0.100:8786
```

The workers connect to the scheduler, which then sets up a long-running network connection back to the worker. The workers will learn the location of other workers from the scheduler.

### Handling Ports

The scheduler and workers both need to accept TCP connections on an open port. By default the scheduler binds to port 8786 and the worker binds to a random open port. If you are behind a firewall then you may have to open particular ports or tell Dask to listen on particular ports with the `--port` and `--worker-port` keywords.:

```
dask-scheduler --port 8000
dask-worker --bokeh-port 8000 --nanny-port 8001
```

## Nanny Processes

Dask workers are run within a Nanny process that monitors the worker process and restarts it if necessary.

## Diagnostic Web Servers

Additionally Dask schedulers and workers host interactive diagnostic web servers using [Bokeh](#). These are optional, but generally useful to users. The diagnostic server on the scheduler is particularly valuable, and is served on port 8787 by default (configurable with the `--bokeh-port` keyword).

- More information about relevant ports is available by looking at the help pages with `dask-scheduler --help` and `dask-worker --help`

## Automated Tools

There are various mechanisms to deploy these executables on a cluster, ranging from manually SSH-ing into all of the machines to more automated systems like SGE/SLURM/Torque or Yarn/Mesos. Additionally, cluster SSH tools exist to send the same commands to many machines. We recommend searching online for “cluster ssh” or “cssh”..

## API

These may be out-dated. We recommend referring to the `--help` text of your installed version.

## dask-scheduler

```
$ dask-scheduler --help
Usage: dask-scheduler [OPTIONS]

Options:
  --host TEXT                URI, IP or hostname of this server
  --port INTEGER             Serving port
  --interface TEXT          Preferred network interface like 'eth0' or 'ib0'
  --tls-ca-file PATH        CA cert(s) file for TLS (in PEM format)
  --tls-cert PATH           certificate file for TLS (in PEM format)
  --tls-key PATH            private key file for TLS (in PEM format)
  --bokeh-port INTEGER      Bokeh port for visual diagnostics
  --bokeh / --no-bokeh     Launch Bokeh Web UI [default: True]
  --show / --no-show       Show web UI
  --bokeh-whitelist TEXT    IP addresses to whitelist for bokeh.
  --bokeh-prefix TEXT       Prefix for the bokeh app
  --use-xheaders BOOLEAN   User xheaders in bokeh app for ssl termination in
                           header [default: False]
  --pid-file TEXT           File to write the process PID
  --scheduler-file TEXT     File to write connection information. This may be a
                           good way to share connection information if your
                           cluster is on a shared network file system.
  --local-directory TEXT    Directory to place scheduler files
  --preload TEXT            Module that should be loaded by each worker process
                           like "foo.bar" or "/path/to/foo.py"
  --help                    Show this message and exit.
```

## dask-worker

```

$ dask-worker --help
Usage: dask-worker [OPTIONS] [SCHEDULER]

Options:
  --tls-ca-file PATH          CA cert(s) file for TLS (in PEM format)
  --tls-cert PATH            certificate file for TLS (in PEM format)
  --tls-key PATH             private key file for TLS (in PEM format)
  --worker-port INTEGER      Serving computation port, defaults to random
  --nanny-port INTEGER       Serving nanny port, defaults to random
  --bokeh-port INTEGER       Bokeh port, defaults to 8789
  --bokeh / --no-bokeh       Launch Bokeh Web UI [default: True]
  --listen-address TEXT      The address to which the worker binds.
                             Example: tcp://0.0.0.0:9000
  --contact-address TEXT     The address the worker advertises to the
                             scheduler for communication with it and other
                             workers. Example: tcp://127.0.0.1:9000
  --host TEXT                Serving host. Should be an ip address that is
                             visible to the scheduler and other workers.
                             See --listen-address and --contact-address if
                             you need different listen and contact
                             addresses. See --interface.
  --interface TEXT          Network interface like 'eth0' or 'ib0'
  --nthreads INTEGER        Number of threads per process.
  --nprocs INTEGER          Number of worker processes. Defaults to one.
  --name TEXT               A unique name for this worker like 'worker-1'
  --memory-limit TEXT       Bytes of memory that the worker can use. This
                             can be an integer (bytes), float (fraction of
                             total system memory), string (like 5GB or
                             5000M), 'auto', or zero for no memory
                             management
  --reconnect / --no-reconnect
                             Reconnect to scheduler if disconnected
  --nanny / --no-nanny      Start workers in nanny process for management
  --pid-file TEXT           File to write the process PID
  --local-directory TEXT    Directory to place worker files
  --resources TEXT          Resources for task constraints like "GPU=2
                             MEM=10e9"
  --scheduler-file TEXT     Filename to JSON encoded scheduler
                             information. Use with dask-scheduler
                             --scheduler-file
  --death-timeout FLOAT     Seconds to wait for a scheduler before closing
  --bokeh-prefix TEXT       Prefix for the bokeh app
  --preload TEXT            Module that should be loaded by each worker
                             process like "foo.bar" or "/path/to/foo.py"
  --help                    Show this message and exit.

```

### 4.2.4 SSH

The convenience script `dask-ssh` opens several SSH connections to your target computers and initializes the network accordingly. You can give it a list of hostnames or IP addresses:

```
$ dask-ssh 192.168.0.1 192.168.0.2 192.168.0.3 192.168.0.4
```

Or you can use normal UNIX grouping:

```
$ dask-ssh 192.168.0.{1,2,3,4}
```

Or you can specify a hostfile that includes a list of hosts:

```
$ cat hostfile.txt
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4

$ dask-ssh --hostfile hostfile.txt
```

The `dask-ssh` utility depends on the `paramiko`:

```
pip install paramiko
```

## 4.2.5 High Performance Computers

### Relevant Machines

This page includes instructions and guidelines when deploying Dask on high performance supercomputers commonly found in scientific and industry research labs. These systems commonly have the following attributes:

1. Some mechanism to launch MPI applications or use job schedulers like SLURM, SGE, TORQUE, LSF, DR-MAA, PBS, or others
2. A shared network file system visible to all machines in the cluster
3. A high performance network interconnect, such as Infiniband
4. Little or no node-local storage

### Using a Shared Network File System and a Job Scheduler

Some clusters benefit from a shared network file system (NFS) and can use this to communicate the scheduler location to the workers:

```
dask-scheduler --scheduler-file /path/to/scheduler.json # writes address to file

dask-worker --scheduler-file /path/to/scheduler.json # reads file for address
dask-worker --scheduler-file /path/to/scheduler.json # reads file for address
```

```
>>> client = Client(scheduler_file='/path/to/scheduler.json')
```

This can be particularly useful when deploying `dask-scheduler` and `dask-worker` processes using a job scheduler like SGE/SLURM/Torque/etc.. Here is an example using SGE's `qsub` command:

```
# Start a dask-scheduler somewhere and write connection information to file
qsub -b y /path/to/dask-scheduler --scheduler-file /home/$USER/scheduler.json

# Start 100 dask-worker processes in an array job pointing to the same file
qsub -b y -t 1-100 /path/to/dask-worker --scheduler-file /home/$USER/scheduler.json
```

Note, the `--scheduler-file` option is *only* valuable if your scheduler and workers share a network file system.

## Using MPI

You can launch a Dask network using `mpirun` or `mpiexec` and the `dask-mpi` command line executable.

```
mpirun --np 4 dask-mpi --scheduler-file /home/$USER/scheduler.json
```

```
from dask.distributed import Client
client = Client(scheduler_file='/path/to/scheduler.json')
```

This depends on the `mpi4py` library. It only uses MPI to start the Dask cluster, and not for inter-node communication. MPI implementations differ. The use of `mpirun --np 4` is specific to the `mpich` MPI implementation installed through `conda` and linked to `mpi4py`

```
conda install mpi4py
```

It is not necessary to use exactly this implementation, but you may want to verify that your `mpi4py` Python library is linked against the proper `mpirun/mpiexec` executable and that the flags used (like `--np 4`) are correct for your system. The system administrator of your cluster should be very familiar with these concerns and able to help.

Run `dask-mpi --help` to see more options for the `dask-mpi` command.

## High Performance Network

Many HPC systems have both standard Ethernet networks as well as high-performance networks capable of increased bandwidth. You can instruct Dask to use the high-performance network interface by using the `--interface` keyword to the `dask-worker`, `dask-scheduler`, or `dask-mpi` commands

```
mpirun --np 4 dask-mpi --scheduler-file /home/$USER/scheduler.json --interface ib0
```

In the code example above we have assumed that your cluster has an Infiniband network interface called `ib0`. You can check this by asking your system administrator or by inspecting the output of `ifconfig`

```
$ ifconfig
lo          Link encap:Local Loopback                # Localhost
           inet addr:127.0.0.1  Mask:255.0.0.0
           inet6 addr: ::1/128 Scope:Host
eth0       Link encap:Ethernet  HWaddr XX:XX:XX:XX:XX:XX  # Ethernet
           inet addr:192.168.0.101
           ...
ib0        Link encap:Infiniband                # Fast InfiniBand
           inet addr:172.42.0.101
```

<https://stackoverflow.com/questions/43881157/how-do-i-use-an-infiniband-network-with-dask>

## No Local Storage

Users often exceed memory limits available to a specific Dask deployment. In normal operation Dask spills excess data to disk. However, in HPC systems the individual compute nodes often lack locally attached storage, preferring instead to store data in a robust high performance network storage solution. As a result when a Dask cluster starts to exceed memory limits its workers can start making many small writes to the remote network file system. This is both inefficient (small writes to a network file system are *much* slower than local storage for this use case) and potentially dangerous to the file system itself.

See [this page](#) for more information on Dask's memory policies. Consider changing the following values to your `~/.dask/config.yaml` file

```
# Fractions of worker memory at which we take action to avoid memory blowup
# Set any of the lower three values to False to turn off the behavior entirely
worker-memory-target: false # don't spill to disk
worker-memory-spill: false # don't spill to disk
worker-memory-pause: 0.80 # fraction at which we pause worker threads
worker-memory-terminate: 0.95 # fraction at which we terminate the worker
```

This stops Dask workers from spilling to disk, and instead relies entirely on mechanisms to stop them from processing when they reach memory limits.

As a reminder, you can set the memory limit for a worker using the `--memory-limit` keyword:

```
dask-mpi ... --memory-limit 10GB
```

Alternatively if you *do* have local storage mounted on your compute nodes you can point Dask workers to use a particular location in your filesystem using the `--local-directory` keyword:

```
dask-mpi ... --local-directory /scratch
```

### Launch Many Small Jobs

HPC job schedulers are optimized for large monolithic jobs with many nodes that all need to run as a group at the same time. Dask jobs can be quite a bit more flexible, workers can come and go without strongly affecting the job. So if we separate our job into many smaller jobs we can often get through the job scheduling queue much more quickly than a typical job. This is particularly valuable when we want to get started right away and interact with a Jupyter notebook session rather than waiting for hours for a suitable allocation block to become free.

So, to get a large cluster quickly we recommend allocating a `dask-scheduler` process on one node with a modest wall time (the intended time of your session) and then allocating many small single-node `dask-worker` jobs with shorter wall times (perhaps 30 minutes) that can easily squeeze into extra space in the job scheduler. As you need more computation you can add more of these single-node jobs or let them expire.

### Use Dask to co-launch a Jupyter server

Dask can help you by launching other services alongside it. For example you can run a Jupyter notebook server on the machine running the `dask-scheduler` process with the following commands

```
from dask.distributed import Client
client = Client(scheduler_file='scheduler.json')

import socket
host = client.run_on_scheduler(socket.gethostname)

def start_jlab(dask_scheduler):
    import subprocess
    proc = subprocess.Popen(['/path/to/jupyter', 'lab', '--ip', host, '--no-browser'])
    dask_scheduler.jlab_proc = proc

client.run_on_scheduler(start_jlab)
```

### Concrete Example with PBS

The Pangeo project maintains instructions on how to deploy Dask on various HPC systems maintained by NCAR using the PBS job scheduler. Their more concrete instructions may not apply to your situation in particular, but it may

be helpful to see a full solution.

- [https://pangeo-data.github.io/pangeo/setup\\_guides/index.html](https://pangeo-data.github.io/pangeo/setup_guides/index.html)

## 4.2.6 Kubernetes (Cloud)

It is easy to launch a Dask cluster and Jupyter notebook server on cloud resources using [Kubernetes](#) and [Helm](#).

This is particularly useful when deploying on Cloud services, like Amazon Web Services, Google Compute Engine, or Microsoft Azure.

### Launch Kubernetes Cluster

This document assumes that you have a Kubernetes cluster and Helm installed.

If this is not the case then you might consider setting up a Kubernetes cluster either on one of the common cloud providers like Google, Amazon, or Microsoft's. We recommend the first part of the documentation in the guide [Zero to JupyterHub](#) that focuses on Kubernetes and Helm. You do not need to follow all of these instructions. JupyterHub is not necessary to deploy Dask:

- [Creating a Kubernetes Cluster](#)
- [Setting up Helm](#)

Alternatively you may want to experiment with Kubernetes locally using [Minikube](#).

### Helm Install Dask

Dask maintains a Helm repository at <https://dask.github.io/helm-chart>. You can add this to your local helm installation as follows (you should have installed Helm as part of the instructions above):

```
helm repo add dask https://dask.github.io/helm-chart
helm repo update
```

Now you can launch Dask on your Kubernetes cluster using the Dask [Helm](#) chart:

```
helm install dask/dask
```

This deploys a `dask-scheduler`, several `dask-worker` processes, and also a Jupyter server.

### Verify Deployment

This might make a minute to deploy. You can check on the status with `kubectl`:

```
kubectl get pods
kubectl get services

$ kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
bald-eel-jupyter-924045334-twtxd    0/1     ContainerCreating  0           1m
bald-eel-scheduler-3074430035-cn1dt 1/1     Running            0           1m
bald-eel-worker-3032746726-202jt    1/1     Running            0           1m
bald-eel-worker-3032746726-b8nqg    1/1     Running            0           1m
bald-eel-worker-3032746726-d0chx    0/1     ContainerCreating  0           1m

$ kubectl get services
```

NAME	AGE	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
bald-eel-jupyter	2m	LoadBalancer	10.11.247.201	35.226.183.149	80:30173/TCP
bald-eel-scheduler	2m	LoadBalancer	10.11.245.241	35.202.201.129	8786:31166/TCP, 80:31626/TCP
kubernetes	48m	ClusterIP	10.11.240.1	<none>	443/TCP

You can use the addresses under `EXTERNAL-IP` to connect to your now-running Jupyter and Dask systems.

Notice the name `bald-eel`. This is the name that Helm has given to your particular deployment of Dask. You could, for example, have multiple Dask-and-Jupyter clusters running at once and each would be given a different name. You will use this name to refer to your deployment in the future. You can list all active helm deployments with:

```
helm list
```

NAME	REVISION	UPDATED	STATUS	CHART
bald-eel	1	Wed Dec 6 11:19:54 2017	DEPLOYED	dask-0.1.

## Connect to Dask and Jupyter

When we ran `kubectl get services` we saw some externally visible IPs

```
mrocklin@pangeo-181919:~$ kubectl get services
```

NAME	AGE	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
bald-eel-jupyter	2m	LoadBalancer	10.11.247.201	35.226.183.149	80:30173/TCP
bald-eel-scheduler	2m	LoadBalancer	10.11.245.241	35.202.201.129	8786:31166/TCP, 80:31626/TCP
kubernetes	48m	ClusterIP	10.11.240.1	<none>	443/TCP

We can navigate to these from any web browser. One is the Dask diagnostic dashboard. The other is the Jupyter server. You can log into the Jupyter notebook server with the password, `dask`.

You can create a notebook and create a Dask client from there. The `DASK_SCHEDULER_ADDRESS` environment variable has been populated with the address of the Dask scheduler. This is available in Python in the `config` dictionary.

```
>>> from dask.distributed import Client, config
>>> config['scheduler-address']
'bal-eel-scheduler:8786'
```

Although you don't need to use this address, the Dask client will find this variable automatically.

## Configure Environment

By default the Helm deployment launches three workers using two cores each and a standard conda environment. We can customize this environment by creating a small yml file that implements a subset of the values in the `dask helm` chart `values.yml` file

For example we can increase the number of workers, and include extra conda and pip packages to install on the both the workers and Jupyter server (these two environments should be matched).

```
# config.yaml

worker:
  replicas: 8
  limits:
    cpu: 2
    memory: 7.5 GiB
  pipPackages: >-
    git+https://github.com/gcsfs/gcsfs.git
    git+https://github.com/xarray/xarray.git
  condaPackages: >-
    -c conda-forge
    zarr
    blosc

# We want to keep the same packages on the worker and jupyter environments
jupyter:
  pipPackages: >-
    git+https://github.com/gcsfs/gcsfs.git
    git+https://github.com/xarray/xarray.git
  condaPackages: >-
    -c conda-forge
    zarr
    blosc
```

This config file overrides configuration for number and size of workers and the conda and pip packages installed on the worker and Jupyter containers. In general we will want to make sure that these two software environments match.

Update your deployment to use this configuration file. Note that *you will not use helm install* for this stage. That would create a *new* deployment on the same Kubernetes cluster. Instead you will upgrade your existing deployment by using the current name:

```
helm upgrade bald-eel dask/dask -f config.yaml
```

This will update those containers that need to be updated. It may take a minute or so.

As a reminder, you can list the names of deployments you have using `helm list`

## Check status and logs

For standard issues you should be able to see worker status and logs using the Dask dashboard (in particular see the worker links from the `info/` page). However if your workers aren't starting you can check on the status of pods and their logs with the following commands

```
kubectl get pods
kubectl logs <PODNAME>
```

```
mrocklin@pangeo-181919:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
bald-eel-jupyter-3805078281-n1qk2   1/1     Running   0           18m
bald-eel-scheduler-3074430035-cn1dt 1/1     Running   0           58m
bald-eel-worker-1931881914-1q09p    1/1     Running   0           18m
bald-eel-worker-1931881914-856mm    1/1     Running   0           18m
bald-eel-worker-1931881914-9lgzb    1/1     Running   0           18m
```

```
bald-eel-worker-1931881914-bdn2c      1/1      Running  0          16m
bald-eel-worker-1931881914-jq70m     1/1      Running  0          17m
bald-eel-worker-1931881914-qsgj7     1/1      Running  0          18m
bald-eel-worker-1931881914-s2phd     1/1      Running  0          17m
bald-eel-worker-1931881914-srmmg     1/1      Running  0          17m

mrocklin@pangeo-181919:~$ kubectl logs bald-eel-worker-1931881914-856mm
EXTRA_CONDA_PACKAGES environment variable found.  Installing.
Fetching package metadata .....
Solving package specifications: .
Package plan for installation in environment /opt/conda/envs/dask:
The following NEW packages will be INSTALLED:
  fasteners: 0.14.1-py36_2 conda-forge
  monotonic: 1.3-py36_0 conda-forge
  zarr:      2.1.4-py36_0 conda-forge
Proceed ([y]/n)?
monotonic-1.3- 100% |#####| Time: 0:00:00 11.16 MB/s
fasteners-0.14 100% |#####| Time: 0:00:00 576.56 kB/s
...
```

### Delete Helm deployment

You can always delete a helm deployment using its name:

```
helm delete bald-eel
```

Note that this does not destroy any clusters that you may have allocated on a Cloud service, you will need to delete those explicitly.

### Avoid the Jupyter Server

Sometimes you do not need to run a Jupyter server alongside your Dask cluster. A simple way to avoid the extra pod is to set `replicas: 0` within your `config.yaml` file under the `jupyter` section.

```
jupyter:
  replicas: 0
```

## 4.2.7 Python API (advanced)

In some rare cases experts may want to create `Scheduler` and `Worker` objects explicitly in Python manually. This is often necessary when making tools to automatically deploy Dask in custom settings.

However, often it is sufficient to rely on the *Dask command line interface*.

### Scheduler

Start the Scheduler, provide the listening port (defaults to 8786) and Tornado `IOLoop` (defaults to `IOLoop.current()`)

```
from distributed import Scheduler
from tornado.ioloop import IOLoop
from threading import Thread
```

```
s = Scheduler()
s.start('tcp://:8786') # Listen on TCP port 8786

loop = IOLoop.current()
loop.start()
```

Alternatively, you may want the IOLoop and scheduler to run in a separate thread. In that case you would replace the `loop.start()` call with the following:

```
t = Thread(target=loop.start, daemon=True)
t.start()
```

## Worker

On other nodes start worker processes that point to the URL of the scheduler.

```
from distributed import Worker
from tornado.ioloop import IOLoop
from threading import Thread

w = Worker('tcp://127.0.0.1:8786')
w.start() # choose randomly assigned port

loop = IOLoop.current()
loop.start()
```

Alternatively, replace `Worker` with `Nanny` if you want your workers to be managed in a separate process by a local nanny process. This allows workers to restart themselves in case of failure, provides some additional monitoring, and is useful when coordinating many workers that should live in different processes to avoid the GIL.

## 4.3 Use Cases

Dask is a versatile tool that supports a variety of workloads. This page contains brief and illustrative examples for how people use Dask in practice. This page emphasizes breadth and hopefully inspires readers to find new ways that Dask can serve them beyond their original intent.

### 4.3.1 Overview

Dask use cases can be roughly divided in the following two categories:

1. Large NumPy/Pandas/Lists with `dask.array`, `dask.dataframe`, `dask.bag` to analyze large datasets with familiar techniques. This is similar to Databases, `Spark`, or big array libraries.
2. Custom task scheduling. You submit a graph of functions that depend on each other for custom workloads. This is similar to `Luigi`, `Airflow`, `Celery`, or `Makefiles`.

Most people today approach Dask assuming it is a framework like `Spark`, designed for the first use case around large collections of uniformly shaped data. However, many of the more productive and novel use cases fall into the second category, using Dask to parallelize custom workflows.

Dask compute environments can be divided into the following two categories:

1. Single machine parallelism with threads or processes: The Dask single-machine scheduler leverages the full CPU power of a laptop or a large workstation and changes the space limitation from “fits in memory” to “fits on disk”. This scheduler is simple to use and doesn’t have the computational or conceptual overhead of most “big data” systems.
2. Distributed cluster parallelism on multiple nodes: The Dask distributed scheduler coordinates the actions of multiple machines on a cluster. It scales anywhere from a single machine to a thousand machines, but not significantly beyond.

The single machine scheduler is useful to more individuals (more people have personal laptops than have access to clusters) and probably accounts for 80+% of the use of Dask today. The distributed machine scheduler is useful to larger organizations like universities, research labs, or private companies.

Below we give specific examples of how people use Dask. We start with large NumPy/Pandas/List examples because they’re somewhat more familiar to people looking at “big data” frameworks. We then follow with custom scheduling examples, which tend to be applicable more often, and are arguably a bit more interesting.

### 4.3.2 Collection Examples

Dask contains large parallel collections for n-dimensional arrays (similar to NumPy), dataframes (similar to Pandas), and lists (similar to PyToolz or PySpark).

#### On disk arrays

Scientists studying the earth have 10GB to 100GB of regularly gridded weather data on their laptop’s hard drive stored as many individual HDF5 or NetCDF files. They use `dask.array` to treat this stack of HDF5 or NetCDF files as a single NumPy array (or a collection of NumPy arrays with the `XArray` project). They slice, perform reductions, perform seasonal averaging etc. all with straight Numpy syntax. These computations take a few minutes (reading 100GB from disk is somewhat slow) but previously infeasible computations become convenient from the comfort of a personal laptop.

It’s not so much parallel computing that is valuable here but rather the ability to comfortably compute on larger-than-memory data without special hardware.

```
import h5py
dataset = h5py.File('myfile.hdf5')['/x']

import dask.array as da
x = da.from_array(dataset, chunks=dataset.chunks)

y = x[::10] - x.mean(axis=0)
y.compute()
```

#### Directory of CSV or tabular HDF files

Analysts studying time series data have a large directory of CSV, HDF, or otherwise formatted tabular files. They usually use `Pandas` for this kind of data but either the volume is too large or dealing with a large number of files is confusing. They use `dask.dataframe` to logically wrap all of these different files into one logical dataframe that is built on demand to save space. Most of their Pandas workflow is the same (Dask.dataframe is a subset of Pandas) so they switch from Pandas to Dask.dataframe and back easily without significantly changing their code.

```
import dask.dataframe as dd
```

```
df = dd.read_csv('data/2016-*.csv', parse_dates=['timestamp'])
df.groupby(df.timestamp.dt.hour).value.mean().compute()
```

### Directory of CSV files on HDFS

The same analyst as above uses `dask.dataframe` with the `dask.distributed` scheduler to analyze terabytes of data on their institution's Hadoop cluster straight from Python. This uses the `HDFS3` Python library for HDFS management

This solution is particularly attractive because it stays within the Python ecosystem, and uses the speed and algorithm set of `Pandas`, a tool with which the analyst is already very comfortable.

```
from dask.distributed import Client
client = Client('cluster-address:8786')

import dask.dataframe as dd
df = dd.read_csv('hdfs://data/2016-*.csv', parse_dates=['timestamp'])
df.groupby(df.timestamp.dt.hour).value.mean().compute()
```

### Directories of custom format files

The same analyst has a bunch of files of a custom format not supported by `Dask.dataframe`, or perhaps these files are in a directory structure that encodes important information about his data (such as the date or other metadata.) They use `dask.delayed` to teach `Dask.dataframe` how to load the data and then pass into `dask.dataframe` for tabular algorithms.

- Example Notebook: <https://gist.github.com/mrocklin/e7b7b3a65f2835cda813096332ec73ca>

### JSON data

Data Engineers with click stream data from a website or mechanical engineers with telemetry data from mechanical instruments have large volumes of data in JSON or some other semi-structured format. They use `dask.bag` to manipulate many Python objects in parallel either on their personal machine, where they stream the data through memory or across a cluster.

```
import dask.bag as db
import json

records = db.read_text('data/2015-*.json').map(json.loads)
records.filter(lambda d: d['name'] == 'Alice').pluck('id').frequencies()
```

## 4.3.3 Custom Examples

The large collections (array, dataframe, bag) are wonderful when they fit the application, for example if you want to perform a groupby on a directory of CSV data. However several parallel computing applications don't fit neatly into one of these higher level abstractions. Fortunately, Dask provides a wide variety of ways to parallelize more custom applications. These use the same machinery as the arrays and dataframes, but allow the user to develop custom algorithms specific to their problem.

### Embarrassingly parallel computation

A programmer has a function that they want to run many times on different inputs. Their function and inputs might use arrays or dataframes internally, but conceptually their problem isn't a single large array or dataframe.

They want to run these functions in parallel on their laptop while they prototype but they also intend to eventually use an in-house cluster. They wrap their function in `dask.delayed` and let the appropriate dask scheduler parallelize and load balance the work.

```
def process(data):  
    ...  
    return ...
```

### Normal Sequential Processing:

```
results = [process(x) for x in inputs]
```

### Build Dask Computation:

```
from dask import compute, delayed  
values = [delayed(process)(x) for x in inputs]
```

### Multiple Threads:

```
import dask.threaded  
results = compute(*values, get=dask.threaded.get)
```

### Multiple Processes:

```
import dask.multiprocessing  
results = compute(*values, get=dask.multiprocessing.get)
```

### Distributed Cluster:

```
from dask.distributed import Client  
client = Client("cluster-address:8786")  
results = compute(*values, get=client.get)
```

## Complex dependencies

A financial analyst has many models that depend on each other in a complex web of computations.

```
data = [load(fn) for fn in filenames]  
reference = load_from_database(query)  
  
A = [model_a(x, reference) for x in data]  
B = [model_b(x, reference) for x in data]  
  
roll_A = [roll(A[i], A[i + 1]) for i in range(len(A) - 1)]  
roll_B = [roll(B[i], B[i + 1]) for i in range(len(B) - 1)]  
compare = [compare_ab(a, b) for a, b in zip(A, B)]  
  
results = summarize(compare, roll_A, roll_B)
```

These models are time consuming and need to be run on a variety of inputs and situations. The analyst has his code now as a collection of Python functions and is trying to figure out how to parallelize such a codebase. They use `dask.delayed` to wrap their function calls and capture the implicit parallelism.

```
from dask import compute, delayed  
  
data = [delayed(load)(fn) for fn in filenames]
```

```
reference = delayed(load_from_database)(query)

A = [delayed(model_a)(x, reference) for x in data]
B = [delayed(model_b)(x, reference) for x in data]

roll_A = [delayed(roll)(A[i], A[i + 1]) for i in range(len(A) - 1)]
roll_B = [delayed(roll)(B[i], B[i + 1]) for i in range(len(B) - 1)]
compare = [delayed(compare_ab)(a, b) for a, b in zip(A, B)]

lazy_results = delayed(summarize)(compare, roll_A, roll_B)
```

They then depend on the dask schedulers to run this complex web of computations in parallel.

```
results = compute(lazy_results)
```

They appreciate how easy it was to transition from the experimental code to a scalable parallel version. This code is also easy enough for their teammates to understand easily and extend in the future.

### Algorithm developer

A graduate student in machine learning is prototyping novel parallel algorithms. They are in a situation much like the financial analyst above except that they need to benchmark and profile their computation heavily under a variety of situations and scales. The dask profiling tools (*single machine diagnostics* and *distributed diagnostics*) provide the feedback they need to understand their parallel performance, including how long each task takes, how intense communication is, and their scheduling overhead. They scale their algorithm between 1 and 50 cores on single workstations and then scale out to a cluster running their computation at thousands of cores. They don't have access to an institutional cluster, so instead they use `dask-ec2` to easily provision clusters of varying sizes.

Their algorithm is written the same in all cases, drastically reducing the cognitive load, and letting the readers of their work experiment with their system on their own machines, aiding reproducibility.

### Scikit-Learn or Joblib User

A data scientist wants to scale their machine learning pipeline to run on their cluster to accelerate parameter searches. They already use the `sklearn njobs=` parameter to accelerate their computation on their local computer with `Joblib`. Now they wrap their `sklearn` code with a context manager to parallelize the exact same code across a cluster (also available with `IPyParallel`)

```
import distributed.joblib

with joblib.parallel_backend('distributed',
                             scheduler_host=('192.168.1.100', 8786)):
    result = GridSearchCV( ... ) # normal sklearn code
```

### Academic Cluster Administrator

A system administrator for a university compute cluster wants to enable many researchers to use the available cluster resources, which are currently lying idle. The research faculty and graduate students lack experience with job schedulers and MPI, but are comfortable interacting with Python code through a Jupyter notebook.

Teaching the faculty and graduate students to parallelize software has proven time consuming. Instead the administrator sets up `dask.distributed` on a sandbox allocation of the cluster and broadly publishes the address of the scheduler, pointing researchers to the `dask.distributed quickstart`. Utilization of the cluster climbs steadily over the next week

as researchers are more easily able to parallelize their computations without having to learn foreign interfaces. The administrator is happy because resources are being used without significant hand-holding.

As utilization increases the administrator has a new problem; the shared `dask.distributed` cluster is being overused. The administrator tracks use through Dask diagnostics to identify which users are taking most of the resources. They contact these users and teach them how to `launch` their own `dask.distributed` clusters using the traditional job scheduler on their cluster, making space for more new users in the sandbox allocation.

### Financial Modeling Team

Similar to the case above, a team of modelers working at a financial institution run a complex network of computational models on top of each other. They started using `dask.delayed` individually, as suggested above, but realized that they often perform highly overlapping computations, such as always reading the same data.

Now they decide to use the same Dask cluster collaboratively to save on these costs. Because Dask intelligently hashes computations in a way similar to how Git works, they find that when two people submit similar computations the overlapping part of the computation runs only once.

Ever since working collaboratively on the same cluster they find that their frequently running jobs run much faster, because most of the work is already done by previous users. When they share scripts with colleagues they find that those repeated scripts complete immediately rather than taking several hours.

They are now able to iterate and share data as a team more effectively, decreasing their time to result and increasing their competitive edge.

As this becomes more heavily used on the company cluster they decide to set up an auto-scaling system. They use their dynamic job scheduler (perhaps SGE, LSF, Mesos, or Marathon) to run a single `dask-scheduler` 24/7 and then scale up and down the number of `dask-workers` running on the cluster based on computational load. This solution ends up being more responsive (and thus more heavily used) than their previous attempts to provide institution-wide access to parallel computing but because it responds to load it still acts as a good citizen in the cluster.

### Streaming data engineering

A data engineer responsible for watching a data feed needs to scale out a continuous process. They `combine` `dask.distributed` with `normal Python Queues` to produce a rudimentary but effective stream processing system.

Because `dask.distributed` is elastic, they can scale up or scale down their cluster resources in response to demand.

## 4.4 Examples

### 4.4.1 Array

*Array documentation*

#### 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
```

## Build Custom Dask.Array Function

As discussed in the [array design document](#) to create a dask Array object we 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 NumPy 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.base import tokenize

def eye(n, blocksize):
    chunks = ((blocksize,) * n // blocksize,
              (blocksize,) * n // blocksize)

    name = 'eye-' + tokenize(n, blocksize) # 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 *i*th block along the diagonal of the output is the result of calling `np.diag` on the *i*th block of the input and all other blocks are zero.

```

from dask.base import tokenize

def diag(v):
    """Construct a diagonal array, with ``v`` on the diagonal."""
    assert v.ndim == 1
    chunks = (v.chunks[0], v.chunks[0]) # repeat chunks twice

    name = 'diag-' + tokenize(v) # 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]])

```

- [Blogpost: Distributed NumPy and Image Analysis on a Cluster, January 2017](#)
- Use `Dask.array` to generate task graphs
- Alternating Least Squares for collaborative filtering

## 4.4.2 Bag

*Bag documentation*

### 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: z1-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().flatten().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 `SCHEDULER_IP` and `SCHEDULER_PORT` with the IP address and port of the distributed scheduler:

```
$ dask-scheduler # On the head node
$ dask-worker SCHEDULER_IP:SCHEDULER_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:

```
>>> from dask.distributed import Client, progress
```

Initialize a connection to the distributed executor:

```
>>> client = Client('SCHEDULER_IP:SCHEDULER_PORT')
```

Create a bag from the text files stored in HDFS. This expression will not read data from HDFS until the computation is triggered:

```
>>> import dask.bag as db
>>> b = db.read_text('hdfs:///tmp/enron/*/*')
```

We can write a word count expression using the same bag methods as the local dask example:

```
>>> wordcount = b.str.split().flatten().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 = client.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 `dask.distributed` workers:

```
>>> results = client.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().flatten().frequencies().topk(10, lambda x: x[1])
wordcount.compute()

# Cluster computation with HDFS

from dask.distributed import Client, progress

client = Client('SCHEDULER_IP:SCHEDULER_PORT')

b = db.read_text('hdfs:///tmp/enron/**')
wordcount = b.str.split().flatten().frequencies().topk(10, lambda x: x[1])

future = client.compute(wordcount)
print(future)
progress(future)

results = client.gather(future)
print(results)
```

## 4.4.3 DataFrame

*DataFrame documentation*

### 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
```

## Dataframes from HDF5 files

This section provides working examples of `dask.dataframe` methods to read HDF5 files. HDF5 is a unique technology suite that makes possible the management of large and complex data collections. To learn more about HDF5, visit the [HDF Group Tutorial page](#). For an overview of `dask.dataframe`, its limitations, scope, and use, see the [DataFrame overview section](#).

**Important Note** – `dask.dataframe.read_hdf` uses `pandas.read_hdf`, thereby inheriting its abilities and limitations. See [pandas HDF5 documentation](#) for more information.

## Examples Covered

- Use `dask.dataframe` to:
  1. Create `dask.DataFrame` by loading a specific dataset (key) from a single HDF5 file
  2. Create `dask.DataFrame` from a single HDF5 file with multiple datasets (keys)
  3. Create `dask.DataFrame` by loading multiple HDF5 files with different datasets (keys)

## Generate Example Data

Here is some code to generate sample HDF5 files.

```
import string, json, random
import pandas as pd
import numpy as np

# dict to keep track of hdf5 filename and each key
fileKeys = {}

for i in range(10):
    # randomly pick letter as dataset key
    groupkey = random.choice(list(string.ascii_lowercase))

    # randomly pick a number as hdf5 filename
    filename = 'my' + str(np.random.randint(100)) + '.h5'

    # Make a dataframe; 26 rows, 2 columns
    df = pd.DataFrame({'x': np.random.randint(1, 1000, 26),
                      'y': np.random.randint(1, 1000, 26)},
                      index=list(string.ascii_lowercase))

    # Write hdf5 to current directory
    df.to_hdf(filename, key='/' + groupkey, format='table')
    fileKeys[filename] = groupkey

print(fileKeys) # prints hdf5 filenames and keys for each
```

## Read single dataset from HDF5

The first order of `dask.dataframe` business is creating a `dask.DataFrame` using a single HDF5 file's dataset. The code to accomplish this task is:

```
import dask.dataframe as dd
df = dd.read_hdf('my86.h5', key='/c')
```

## Load multiple datasets from single HDF5 file

Loading multiple datasets from a single file requires a small tweak and use of the wildcard character:

```
import dask.dataframe as dd
df = dd.read_hdf('my86.h5', key='/*')
```

Learn more about `dask.dataframe` methods by visiting the [API documentation](#).

### Create dask DataFrame from multiple HDF5 files

The next example is a natural progression from the previous example (e.g. using a wildcard). Add a wildcard for the `key` and `path` parameters to read multiple files and multiple keys:

```
import dask.dataframe as dd
df = dd.read_hdf('./*.h5', key='/*')
```

These exercises cover the basics of using `dask.dataframe` to work with HDF5 data. For more information on the user functions to manipulate and explore dataframes (visualize, describe, compute, etc.) see [API documentation](#). To explore the other data formats supported by `dask.dataframe`, visit the [section on creating dataframes](#).

- Blogpost: Dataframes on a cluster, January 2017
- Distributed DataFrames on NYCTaxi data
- Build Parallel Algorithms for Pandas
- Simple distributed joins
- Build Dask.dataframes from custom format, feather

## 4.4.4 Delayed

*Delayed documentation*

### 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.

- [Blogpost: Delayed on a cluster, January 2017](#)
- [Blogpost: Dask and Celery, September 2016](#)

- Basic Delayed example
- Build Parallel Algorithms for Pandas
- Build Dask.dataframes from custom format, feather

#### 4.4.5 Distributed Concurrent.futures

Concurrent.futures documentation

- Custom workflows
- Ad Hoc Distributed Random Forests
- Web Servers and Asynchronous task scheduling

#### 4.4.6 Tutorial

A Dask tutorial from July 2015 (fairly old) is available here: <https://github.com/dask/dask-tutorial>

### 4.5 Community

Dask is used and developed by individuals at a variety of institutions. It sits within the broader Python numeric ecosystem commonly referred to as PyData or SciPy.

#### 4.5.1 Discussion

Conversation happens in the following places:

1. **Usage questions** are directed to [Stack Overflow](#) with the `#dask` tag. Dask developers monitor this tag and get e-mails whenever a question is asked.
2. **Bug reports and feature requests** are managed on the [GitHub issue tracker](#)
3. **Chat** occurs on at [gitter.im/dask/dask](https://gitter.im/dask/dask) for general conversation and [gitter.im/dask/dev](https://gitter.im/dask/dev) for developer conversation. Note that because gitter chat is not searchable by future users we discourage usage questions and bug reports on gitter and instead ask people to use Stack Overflow or GitHub.
4. **Weekly developer meeting** happens at Thursday 4pm UTC (11am in New York, 8am in Los Angeles, 12am in Beijing) at <https://appear.in/dask-dev> with meeting notes on a publicly viewable [document](#)

#### 4.5.2 Asking for help

We welcome usage questions and bug reports from all users, even those who are new to using the project. There are a few things you can do to improve the likelihood of quickly getting a good answer.

1. **Ask questions in the right place.** In particular 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 strictly reserved for developer and community discussion.
2. **Create a minimal example.** It is ideal to create [minimal, complete, verifiable examples](#). This significantly reduces the time that answerers spend understanding your situation and so results in higher quality answers more quickly.

### 4.5.3 Paid support

Dask is an open source project that originated at [Anaconda Inc.](#). In addition to the previous options, Anaconda offers paid training and support: <https://www.anaconda.com/support>.

#### 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*
- *Futures*

## 4.6 Array

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

### 4.6.1 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 oceanography, 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 array 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 with unknown shapes do not support all operations
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 array doesn't implement operations like `tolist` that would be very inefficient for larger datasets. Likewise it is very inefficient to iterate over a Dask array with `for` loops.
5. Dask development is driven by immediate need, and so many lesser used functions have not been implemented. Community contributions are encouraged.

### 4.6.2 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 each of these `Dask.delayed` objects into a single-chunk Dask array with `da.from_delayed`. You can then arrange these single-chunk Dask arrays into a larger multiple-chunk Dask array using [concatenation and stacking](#), as described above.

See [documentation on using dask.delayed with collections](#).

## From `Dask.dataframe`

You can create dask arrays from dask dataframes using the `.values` attribute or the `.to_records()` method.

```
>>> x = df.values
>>> x = df.to_records()
```

However these arrays do not have known chunk sizes (`dask.dataframe` does not track the number of rows in each partition) and so some operations like slicing will not operate correctly.

## Interactions with NumPy arrays

`Dask.array` operations will automatically convert NumPy arrays into single-chunk dask arrays

```
>>> x = da.sum(np.ones(5))
>>> x.compute()
5
```

When NumPy and Dask arrays interact the result will be a Dask array. Automatic rechunking rules will generally slice the NumPy array into the appropriate Dask chunk shape

```
>>> x = da.ones(10, chunks=(5,))
>>> y = np.ones(10)
>>> z = x + y
>>> z
dask.array<add, shape=(10,), dtype=float64, chunksize=(5,)>
```

These interactions work not just for NumPy arrays, but for any object that has shape and dtype attributes and implements NumPy slicing syntax.

## 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.

For performance, 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.

## Unknown Chunks

Some arrays have unknown chunk sizes. These are designated using `np.nan` rather than an integer. These arrays support many but not all operations. In particular, operations like slicing are not possible and will result in an error.

```
>>> x.shape
(np.nan, np.nan)

>>> x[0]
ValueError: Array chunk sizes unknown
```

## Chunks Examples

We show of how different inputs for `chunks=` cut up the following array:

```
1 2 3 4 5 6
7 8 9 0 1 2
3 4 5 6 7 8
9 0 1 2 3 4
5 6 7 8 9 0
1 2 3 4 5 6
```

We show how different `chunks=` arguments split the array into different blocks

**chunks=3:** Symmetric blocks of size 3:

```
1 2 3 4 5 6
7 8 9 0 1 2
3 4 5 6 7 8

9 0 1 2 3 4
5 6 7 8 9 0
1 2 3 4 5 6
```

**chunks=2:** Symmetric blocks of size 2:

```
1 2 3 4 5 6
7 8 9 0 1 2

3 4 5 6 7 8
9 0 1 2 3 4

5 6 7 8 9 0
1 2 3 4 5 6
```

**chunks=(3, 2):** Asymmetric but repeated blocks of size (3, 2):

```
1 2 3 4 5 6
7 8 9 0 1 2
3 4 5 6 7 8

9 0 1 2 3 4
5 6 7 8 9 0
1 2 3 4 5 6
```

**chunks=(1, 6):** Asymmetric but repeated blocks of size (1, 6):

```
1 2 3 4 5 6

7 8 9 0 1 2

3 4 5 6 7 8

9 0 1 2 3 4

5 6 7 8 9 0

1 2 3 4 5 6
```

**chunks=((2, 4), (3, 3)):** Asymmetric and non-repeated blocks:

```
1 2 3 4 5 6
7 8 9 0 1 2

3 4 5 6 7 8
9 0 1 2 3 4
5 6 7 8 9 0
1 2 3 4 5 6
```

**chunks=((2, 2, 1, 1), (3, 2, 1)):** Asymmetric and non-repeated blocks:

```

1 2 3 4 5 6
7 8 9 0 1 2

3 4 5 6 7 8
9 0 1 2 3 4

5 6 7 8 9 0

1 2 3 4 5 6

```

### Discussion

The latter examples are rarely provided by users on original data but arise from complex slicing and broadcasting operations. Generally people use the simplest form until they need more complex forms. The choice of chunks should align with the computations you want to do.

For example, if you plan to take out thin slices along the first dimension then you might want to make that dimension skinnier than the others. If you plan to do linear algebra then you might want more symmetric blocks.

## 4.6.3 Store Dask Arrays

### In Memory

If you have a small amount of data, you can call `np.array` or `.compute()` 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])

>>> y.compute()
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, output2]) # doctest: +SKIP
```

## 4.6.4 Plugins

We can run arbitrary user-defined functions on dask.arrays whenever they are constructed. This allows us to build a variety of custom behaviors that improve debugging, user warning, etc.. You can register a list of functions to run on all dask.arrays to the global `array_plugins=` value:

```
>>> def f(x):
...     print(x.nbytes)

>>> with dask.set_options(array_plugins=[f]):
...     x = da.ones((10, 1), chunks=(5, 1))
...     y = x.dot(x.T)
80
80
800
800
```

If the plugin function returns `None` then the input Dask.array will be returned without change. If the plugin function returns something else then that value will be the result of the constructor.

## Examples

### Automatically compute

We may wish to turn some Dask.array code into normal NumPy code. This is useful for example to track down errors immediately that would otherwise be hidden by Dask's lazy semantics.

```
>>> with dask.set_options(array_plugins=[lambda x: x.compute()]):
...     x = da.arange(5, chunks=2)

>>> x # this was automatically converted into a numpy array
array([0, 1, 2, 3, 4])
```

### Warn on large chunks

We may wish to warn users if they are creating chunks that are too large

```
def warn_on_large_chunks(x):
    shapes = list(itertools.product(*x.chunks))
    nbytes = [x.dtype.itemsize * np.prod(shape) for shape in shapes]
    if any(nb > 1e9 for nb in nbytes):
        warnings.warn("Array contains very large chunks")

with dask.set_options(array_plugins=[warn_on_large_chunks]):
    ...
```

## Combine

You can also combine these plugins into a list. They will run one after the other, chaining results through them.

```
with dask.set_options(array_plugins=[warn_on_large_chunks, lambda x: x.compute()]):
    ...
```

### 4.6.5 API

Top level user functions:

<code>all(a[, axis, out, keepdims])</code>	Test whether all array elements along a given axis evaluate to True.
<code>allclose(a, b[, rtol, atol, equal_nan])</code>	Returns True if two arrays are element-wise equal within a tolerance.
<code>angle(x[, deg])</code>	Return the angle of the complex argument.
<code>any(a[, axis, out, keepdims])</code>	Test whether any array element along a given axis evaluates to True.
<code>apply_along_axis(func1d, axis, arr, *args, ...)</code>	Apply a function to 1-D slices along the given axis.
<code>apply_over_axes(func, a, axes)</code>	Apply a function repeatedly over multiple axes.
<code>arange(*args, **kwargs)</code>	Return evenly spaced values from <i>start</i> to <i>stop</i> with step size <i>step</i> .
<code>arccos(x[, out])</code>	Trigonometric inverse cosine, element-wise.
<code>arccosh(x[, out])</code>	Inverse hyperbolic cosine, element-wise.
<code>arcsin(x[, out])</code>	Inverse sine, element-wise.
<code>arcsinh(x[, out])</code>	Inverse hyperbolic sine element-wise.
<code>arctan(x[, out])</code>	Trigonometric inverse tangent, element-wise.
<code>arctan2(x1, x2[, out])</code>	Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.
<code>arctanh(x[, out])</code>	Inverse hyperbolic tangent element-wise.
<code>argmax(a[, axis, out])</code>	Returns the indices of the maximum values along an axis.
<code>argmin(a[, axis, out])</code>	Returns the indices of the minimum values along an axis.
<code>argwhere(a)</code>	Find the indices of array elements that are non-zero, grouped by element.
<code>around(a[, decimals, out])</code>	Evenly round to the given number of decimals.
<code>array(object[, dtype, copy, order, subok, ndmin])</code>	Create an array.
<code>asanyarray(a)</code>	Convert the input to a dask array.
<code>asarray(a)</code>	Convert the input to a dask array.
<code>atleast_1d(*arys)</code>	Convert inputs to arrays with at least one dimension.
<code>atleast_2d(*arys)</code>	View inputs as arrays with at least two dimensions.
<code>atleast_3d(*arys)</code>	View inputs as arrays with at least three dimensions.
<code>bincount(x[, weights, minlength])</code>	Count number of occurrences of each value in array of non-negative ints.
<code>block(arrays[, allow_unknown_chunksizes])</code>	Assemble an nd-array from nested lists of blocks.
<code>broadcast_to(x, shape[, chunks])</code>	Broadcast an array to a new shape.
<code>coarsen(reduction, x, axes[, trim_excess])</code>	Coarsen array by applying reduction to fixed size neighborhoods
<code>ceil(x[, out])</code>	Return the ceiling of the input, element-wise.
<code>choose(a, choices[, out, mode])</code>	Construct an array from an index array and a set of arrays to choose from.

Continued on next page

Table 4.1 – continued from previous page

<code>clip(*args, **kwargs)</code>	Clip (limit) the values in an array.
<code>compress(condition, a[, axis, out])</code>	Return selected slices of an array along given axis.
<code>concatenate(seq[, axis, ...])</code>	Concatenate arrays along an existing axis
<code>conj(x[, out])</code>	Return the complex conjugate, element-wise.
<code>copysign(x1, x2[, out])</code>	Change the sign of x1 to that of x2, element-wise.
<code>corrcoef(x[, y, rowvar, bias, ddof])</code>	Return Pearson product-moment correlation coefficients.
<code>cos(x[, out])</code>	Cosine element-wise.
<code>cosh(x[, out])</code>	Hyperbolic cosine, element-wise.
<code>count_nonzero(a)</code>	Counts the number of non-zero values in the array a.
<code>cov(m[, y, rowvar, bias, ddof, fweights, ...])</code>	Estimate a covariance matrix, given data and weights.
<code>cumprod(a[, axis, dtype, out])</code>	Return the cumulative product of elements along a given axis.
<code>cumsum(a[, axis, dtype, out])</code>	Return the cumulative sum of the elements along a given axis.
<code>deg2rad(x[, out])</code>	Convert angles from degrees to radians.
<code>degrees(x[, out])</code>	Convert angles from radians to degrees.
<code>diag(v[, k])</code>	Extract a diagonal or construct a diagonal array.
<code>diff(a[, n, axis])</code>	Calculate the n-th discrete difference along given axis.
<code>digitize(x, bins[, right])</code>	Return the indices of the bins to which each value in input array belongs.
<code>dot(a, b[, out])</code>	Dot product of two arrays.
<code>dstack(tup)</code>	Stack arrays in sequence depth wise (along third axis).
<code>ediff1d(ary[, to_end, to_begin])</code>	The differences between consecutive elements of an array.
<code>empty</code>	Blocked variant of empty
<code>empty_like(a[, dtype, chunks])</code>	Return a new array with the same shape and type as a given array.
<code>exp(x[, out])</code>	Calculate the exponential of all elements in the input array.
<code>expm1(x[, out])</code>	Calculate $\exp(x) - 1$ for all elements in the array.
<code>eye(N, chunks[, M, k, dtype])</code>	Return a 2-D Array with ones on the diagonal and zeros elsewhere.
<code>fabs(x[, out])</code>	Compute the absolute values element-wise.
<code>fix(*args, **kwargs)</code>	Round to nearest integer towards zero.
<code>flatnonzero(a)</code>	Return indices that are non-zero in the flattened version of a.
<code>flip(m, axis)</code>	Reverse element order along axis.
<code>flipud(m)</code>	Flip array in the up/down direction.
<code>fliplr(m)</code>	Flip array in the left/right direction.
<code>floor(x[, out])</code>	Return the floor of the input, element-wise.
<code>fmax(x1, x2[, out])</code>	Element-wise maximum of array elements.
<code>fmin(x1, x2[, out])</code>	Element-wise minimum of array elements.
<code>fmod(x1, x2[, out])</code>	Return the element-wise remainder of division.
<code>frexp(x[, out1, out2])</code>	Decompose the elements of x into mantissa and twos exponent.
<code>fromfunction(function, shape, **kwargs)</code>	Construct an array by executing a function over each coordinate.
<code>frompyfunc(func, nin, nout)</code>	Takes an arbitrary Python function and returns a Numpy ufunc.
<code>full</code>	Blocked variant of full
<code>full_like(a, fill_value[, dtype, chunks])</code>	Return a full array with the same shape and type as a given array.
<code>histogram(a[, bins, range, normed, weights, ...])</code>	Blocked variant of numpy.histogram.

Continued on next page

Table 4.1 – continued from previous page

<code>hstack(tup)</code>	Stack arrays in sequence horizontally (column wise).
<code>hypot(x1, x2[, out])</code>	Given the “legs” of a right triangle, return its hypotenuse.
<code>imag(*args, **kwargs)</code>	Return the imaginary part of the elements of the array.
<code>indices(dimensions[, dtype, chunks])</code>	Implements NumPy’s <code>indices</code> for Dask Arrays.
<code>insert(arr, obj, values[, axis])</code>	Insert values along the given axis before the given indices.
<code>isclose(a, b[, rtol, atol, equal_nan])</code>	Returns a boolean array where two arrays are element-wise equal within a tolerance.
<code>iscomplex(*args, **kwargs)</code>	Returns a bool array, where True if input element is complex.
<code>isfinite(x[, out])</code>	Test element-wise for finiteness (not infinity or not Not a Number).
<code>isinf(x[, out])</code>	Test element-wise for positive or negative infinity.
<code>isnan(x[, out])</code>	Test element-wise for NaN and return result as a boolean array.
<code>isnull(values)</code>	<code>pandas.isnull</code> for dask arrays
<code>isreal(*args, **kwargs)</code>	Returns a bool array, where True if input element is real.
<code>ldexp(x1, x2[, out])</code>	Returns $x1 * 2^{x2}$ , element-wise.
<code>linspace(start, stop[, num, chunks, dtype])</code>	Return <i>num</i> evenly spaced values over the closed interval <i>[start, stop]</i> .
<code>log(x[, out])</code>	Natural logarithm, element-wise.
<code>log10(x[, out])</code>	Return the base 10 logarithm of the input array, element-wise.
<code>log1p(x[, out])</code>	Return the natural logarithm of one plus the input array, element-wise.
<code>log2(x[, out])</code>	Base-2 logarithm of <i>x</i> .
<code>logaddexp(x1, x2[, out])</code>	Logarithm of the sum of exponentiations of the inputs.
<code>logaddexp2(x1, x2[, out])</code>	Logarithm of the sum of exponentiations of the inputs in base-2.
<code>logical_and(x1, x2[, out])</code>	Compute the truth value of <i>x1</i> AND <i>x2</i> element-wise.
<code>logical_not(x[, out])</code>	Compute the truth value of NOT <i>x</i> element-wise.
<code>logical_or(x1, x2[, out])</code>	Compute the truth value of <i>x1</i> OR <i>x2</i> element-wise.
<code>logical_xor(x1, x2[, out])</code>	Compute the truth value of <i>x1</i> XOR <i>x2</i> , element-wise.
<code>map_blocks(func, *args, **kwargs)</code>	Map a function across all blocks of a dask array.
<code>matmul(a, b[, out])</code>	Matrix product of two arrays.
<code>max(a[, axis, out, keepdims])</code>	Return the maximum of an array or maximum along an axis.
<code>maximum(x1, x2[, out])</code>	Element-wise maximum of array elements.
<code>mean(a[, axis, dtype, out, keepdims])</code>	Compute the arithmetic mean along the specified axis.
<code>meshgrid(*xi, **kwargs)</code>	Return coordinate matrices from coordinate vectors.
<code>min(a[, axis, out, keepdims])</code>	Return the minimum of an array or minimum along an axis.
<code>minimum(x1, x2[, out])</code>	Element-wise minimum of array elements.
<code>modf(x[, out1, out2])</code>	Return the fractional and integral parts of an array, element-wise.
<code>moment(a, order[, axis, dtype, keepdims, ...])</code>	
<code>nanargmax(x, axis, **kwargs)</code>	
<code>nanargmin(x, axis, **kwargs)</code>	
<code>nancumprod(a[, axis, dtype, out])</code>	Return the cumulative product of array elements over a given axis treating Not a Numbers (NaNs) as one.
<code>nancumsum(a[, axis, dtype, out])</code>	Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.

Continued on next page

Table 4.1 – continued from previous page

<code>nanmax(a[, axis, out, keepdims])</code>	Return the maximum of an array or maximum along an axis, ignoring any NaNs.
<code>nanmean(a[, axis, dtype, out, keepdims])</code>	Compute the arithmetic mean along the specified axis, ignoring NaNs.
<code>nanmin(a[, axis, out, keepdims])</code>	Return minimum of an array or minimum along an axis, ignoring any NaNs.
<code>nanprod(a[, axis, dtype, out, keepdims])</code>	Return the product of array elements over a given axis treating Not a Numbers (NaNs) as zero.
<code>nanstd(a[, axis, dtype, out, ddof, keepdims])</code>	Compute the standard deviation along the specified axis, while ignoring NaNs.
<code>nansum(a[, axis, dtype, out, keepdims])</code>	Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
<code>nanvar(a[, axis, dtype, out, ddof, keepdims])</code>	Compute the variance along the specified axis, while ignoring NaNs.
<code>nextafter(x1, x2[, out])</code>	Return the next floating-point value after x1 towards x2, element-wise.
<code>nonzero(a)</code>	Return the indices of the elements that are non-zero.
<code>notnull(values)</code>	<code>pandas.notnull</code> for dask arrays
<code>ones</code>	Blocked variant of ones
<code>ones_like(a[, dtype, chunks])</code>	Return an array of ones with the same shape and type as a given array.
<code>percentile(a, q[, interpolation])</code>	Approximate percentile of 1-D array
<code>prod(a[, axis, dtype, out, keepdims])</code>	Return the product of array elements over a given axis.
<code>ptp(a[, axis, out])</code>	Range of values (maximum - minimum) along an axis.
<code>rad2deg(x[, out])</code>	Convert angles from radians to degrees.
<code>radians(x[, out])</code>	Convert angles from degrees to radians.
<code>ravel(a[, order])</code>	Return a contiguous flattened array.
<code>real(*args, **kwargs)</code>	Return the real part of the elements of the array.
<code>rechunk(x, chunks[, threshold, block_size_limit])</code>	Convert blocks in dask array x for new chunks.
<code>repeat(a, repeats[, axis])</code>	Repeat elements of an array.
<code>reshape(x, shape)</code>	Reshape array to new shape
<code>result_type(*arrays_and_dtypes)</code>	Returns the type that results from applying the NumPy type promotion rules to the arguments.
<code>rint(x[, out])</code>	Round elements of the array to the nearest integer.
<code>roll(a, shift[, axis])</code>	Roll array elements along a given axis.
<code>round(a[, decimals, out])</code>	Round an array to the given number of decimals.
<code>sign(x[, out])</code>	Returns an element-wise indication of the sign of a number.
<code>signbit(x[, out])</code>	Returns element-wise True where signbit is set (less than zero).
<code>sin(x[, out])</code>	Trigonometric sine, element-wise.
<code>sinh(x[, out])</code>	Hyperbolic sine, element-wise.
<code>sqrt(x[, out])</code>	Return the positive square-root of an array, element-wise.
<code>square(x[, out])</code>	Return the element-wise square of the input.
<code>squeeze(a[, axis])</code>	Remove single-dimensional entries from the shape of an array.
<code>stack(seq[, axis])</code>	Stack arrays along a new axis
<code>std(a[, axis, dtype, out, ddof, keepdims])</code>	Compute the standard deviation along the specified axis.
<code>sum(a[, axis, dtype, out, keepdims])</code>	Sum of array elements over a given axis.
<code>take(a, indices[, axis, out, mode])</code>	Take elements from an array along an axis.
<code>tan(x[, out])</code>	Compute tangent element-wise.
<code>tanh(x[, out])</code>	Compute hyperbolic tangent element-wise.

Continued on next page

Table 4.1 – continued from previous page

<code>tensor_dot(a, b[, axes])</code>	Compute tensor dot product along specified axes for arrays $\geq$ 1-D.
<code>tile(A, reps)</code>	Construct an array by repeating A the number of times given by reps.
<code>topk(k, x)</code>	The top k elements of an array
<code>transpose(a[, axes])</code>	Permute the dimensions of an array.
<code>tril(m[, k])</code>	Lower triangle of an array with elements above the $k$ -th diagonal zeroed.
<code>triu(m[, k])</code>	Upper triangle of an array with elements above the $k$ -th diagonal zeroed.
<code>trunc(x[, out])</code>	Return the truncated value of the input, element-wise.
<code>unique(ar[, return_index, return_inverse, ...])</code>	Find the unique elements of an array.
<code>var(a[, axis, dtype, out, ddof, keepdims])</code>	Compute the variance along the specified axis.
<code>vdot(a, b)</code>	Return the dot product of two vectors.
<code>vnorm(a[, ord, axis, dtype, keepdims, ...])</code>	Vector norm
<code>vstack(tup)</code>	Stack arrays in sequence vertically (row wise).
<code>where(condition, [x, y])</code>	Return elements, either from $x$ or $y$ , depending on <i>condition</i> .
<code>zeros</code>	Blocked variant of zeros
<code>zeros_like(a[, dtype, chunks])</code>	Return an array of zeros with the same shape and type as a given array.

## Fast Fourier Transforms

<code>fft.fft_wrap(fft_func[, kind, dtype])</code>	Wrap 1D complex FFT functions
<code>fft.fft(a[, n, axis])</code>	Wrapping of <code>numpy.fft.fftpack.fft</code>
<code>fft.fft2(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.fft2</code>
<code>fft.fftn(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.fftn</code>
<code>fft.ifft(a[, n, axis])</code>	Wrapping of <code>numpy.fft.fftpack.ifft</code>
<code>fft.ifft2(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.ifft2</code>
<code>fft.ifftn(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.ifftn</code>
<code>fft.rfft(a[, n, axis])</code>	Wrapping of <code>numpy.fft.fftpack.rfft</code>
<code>fft.rfft2(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.rfft2</code>
<code>fft.rfftn(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.rfftn</code>
<code>fft.irfft(a[, n, axis])</code>	Wrapping of <code>numpy.fft.fftpack.irfft</code>
<code>fft.irfft2(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.irfft2</code>
<code>fft.irfftn(a[, s, axes])</code>	Wrapping of <code>numpy.fft.fftpack.irfftn</code>
<code>fft.hfft(a[, n, axis])</code>	Wrapping of <code>numpy.fft.fftpack.hfft</code>
<code>fft.ihfft(a[, n, axis])</code>	Wrapping of <code>numpy.fft.fftpack.ihfft</code>
<code>fft.fftfreq(n[, d])</code>	Return the Discrete Fourier Transform sample frequencies.
<code>fft.rfftfreq(n[, d])</code>	Return the Discrete Fourier Transform sample frequencies (for usage with <code>rfft</code> , <code>irfft</code> ).
<code>fft.fftshift(x[, axes])</code>	Shift the zero-frequency component to the center of the spectrum.
<code>fft.ifftshift(x[, axes])</code>	The inverse of <code>fftshift</code> .

## Linear Algebra

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

## Masked Arrays

<code>ma.filled(a[, fill_value])</code>	Return input as an array with masked data replaced by a fill value.
<code>ma.fix_invalid(a[, mask, copy, fill_value])</code>	Return input with invalid data masked and replaced by a fill value.
<code>ma.getdata(a[, subok])</code>	Return the data of a masked array as an ndarray.
<code>ma.getmaskarray(arr)</code>	Return the mask of a masked array, or full boolean array of False.
<code>ma.masked_array([data, mask, dtype, copy, ...])</code>	An array class with possibly masked values.
<code>ma.masked_equal(x, value[, copy])</code>	Mask an array where equal to a given value.
<code>ma.masked_greater(x, value[, copy])</code>	Mask an array where greater than a given value.
<code>ma.masked_greater_equal(x, value[, copy])</code>	Mask an array where greater than or equal to a given value.
<code>ma.masked_inside(x, v1, v2[, copy])</code>	Mask an array inside a given interval.
<code>ma.masked_invalid(a[, copy])</code>	Mask an array where invalid values occur (NaNs or infs).
<code>ma.masked_less(x, value[, copy])</code>	Mask an array where less than a given value.
<code>ma.masked_less_equal(x, value[, copy])</code>	Mask an array where less than or equal to a given value.
<code>ma.masked_not_equal(x, value[, copy])</code>	Mask an array where <i>not</i> equal to a given value.
<code>ma.masked_outside(x, v1, v2[, copy])</code>	Mask an array outside a given interval.
<code>ma.masked_values(x, value[, rtol, atol, ...])</code>	Mask using floating point equality.
<code>ma.masked_where(condition, a[, copy])</code>	Mask an array where a condition is met.
<code>ma.set_fill_value(a, fill_value)</code>	Set the filling value of $a$ , if $a$ is a masked array.

## Random

<code>random.beta(a, b[, size])</code>	Draw samples from a Beta distribution.
<code>random.binomial(n, p[, size])</code>	Draw samples from a binomial distribution.
<code>random.chisquare(df[, size])</code>	Draw samples from a chi-square distribution.
<code>random.choice(a[, size, replace, p])</code>	Generates a random sample from a given 1-D array
<code>random.exponential([scale, size])</code>	Draw samples from an exponential distribution.
<code>random.f(dfnum, dfden[, size])</code>	Draw samples from an F distribution.
<code>random.gamma(shape[, scale, size])</code>	Draw samples from a Gamma distribution.

Continued on next page

Table 4.5 – continued from previous page

<code>random.geometric(p[, size])</code>	Draw samples from the geometric distribution.
<code>random.gumbel([loc, scale, size])</code>	Draw samples from a Gumbel distribution.
<code>random.hypergeometric(ngood, nbad, nsample)</code>	Draw samples from a Hypergeometric distribution.
<code>random.laplace([loc, scale, size])</code>	Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).
<code>random.logistic([loc, scale, size])</code>	Draw samples from a logistic distribution.
<code>random.lognormal([mean, sigma, size])</code>	Draw samples from a log-normal distribution.
<code>random.logseries(p[, size])</code>	Draw samples from a logarithmic series distribution.
<code>random.negative_binomial(n, p[, size])</code>	Draw samples from a negative binomial distribution.
<code>random.noncentral_chisquare(df, nonc[, size])</code>	Draw samples from a noncentral chi-square distribution.
<code>random.noncentral_f(dfnum, dfden, nonc[, size])</code>	Draw samples from the noncentral F distribution.
<code>random.normal([loc, scale, size])</code>	Draw random samples from a normal (Gaussian) distribution.
<code>random.pareto(a[, size])</code>	Draw samples from a Pareto II or Lomax distribution with specified shape.
<code>random.poisson([lam, size])</code>	Draw samples from a Poisson distribution.
<code>random.power(a[, size])</code>	Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
<code>random.random([size])</code>	Return random floats in the half-open interval [0.0, 1.0).
<code>random.random_sample([size])</code>	Return random floats in the half-open interval [0.0, 1.0).
<code>random.rayleigh([scale, size])</code>	Draw samples from a Rayleigh distribution.
<code>random.standard_cauchy([size])</code>	Draw samples from a standard Cauchy distribution with mode = 0.
<code>random.standard_exponential([size])</code>	Draw samples from the standard exponential distribution.
<code>random.standard_gamma(shape[, size])</code>	Draw samples from a standard Gamma distribution.
<code>random.standard_normal([size])</code>	Draw samples from a standard Normal distribution (mean=0, stdev=1).
<code>random.standard_t(df[, size])</code>	Draw samples from a standard Student's t distribution with <i>df</i> degrees of freedom.
<code>random.triangular(left, mode, right[, size])</code>	Draw samples from the triangular distribution.
<code>random.uniform([low, high, size])</code>	Draw samples from a uniform distribution.
<code>random.vonmises(mu, kappa[, size])</code>	Draw samples from a von Mises distribution.
<code>random.wald(mean, scale[, size])</code>	Draw samples from a Wald, or inverse Gaussian, distribution.
<code>random.weibull(a[, size])</code>	Draw samples from a Weibull distribution.
<code>random.zipf(a[, size])</code>	Standard distributions

## Stats

<code>stats.ttest_ind(a, b[, axis, equal_var])</code>	Calculates the T-test for the means of TWO INDEPENDENT samples of scores.
<code>stats.ttest_1samp(a, popmean[, axis, nan_policy])</code>	Calculates the T-test for the mean of ONE group of scores.
<code>stats.ttest_rel(a, b[, axis, nan_policy])</code>	Calculates the T-test on TWO RELATED samples of scores, a and b.
<code>stats.chisquare(f_obs[, f_exp, ddof, axis])</code>	Calculates a one-way chi square test.
<code>stats.power_divergence(f_obs[, f_exp, ddof, ...])</code>	Cressie-Read power divergence statistic and goodness of fit test.
<code>stats.skew(a[, axis, bias, nan_policy])</code>	Computes the skewness of a data set.

Continued on next page

Table 4.6 – continued from previous page

<code>stats.skewtest(a[, axis, nan_policy])</code>	Tests whether the skew is different from the normal distribution.
<code>stats.kurtosis(a[, axis, fisher, bias, ...])</code>	Computes the kurtosis (Fisher or Pearson) of a dataset.
<code>stats.kurtosistest(a[, axis, nan_policy])</code>	Tests whether a dataset has normal kurtosis
<code>stats.normaltest(a[, axis, nan_policy])</code>	Tests whether a sample differs from a normal distribution.
<code>stats.f_oneway(*args)</code>	Performs a 1-way ANOVA.
<code>stats.moment(a[, moment, axis, nan_policy])</code>	Calculates the nth moment about the mean for a sample.

## Image Support

<code>image.imread(filename[, imread, preprocess])</code>	Read a stack of images into a dask array
---	--

## Slightly Overlapping Ghost Computations

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

## Create and Store Arrays

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

## Internal functions

<code>atop(func, out_ind, *args, **kwargs)</code>	Tensor operation: Generalized inner and outer products
<code>top(func, output, out_indices, ...)</code>	Tensor operation

## Other functions

`dask.array.from_array(x, chunks, name=None, lock=False, asarray=True, fancy=True, getitem=None)`

Create dask array from something that looks like an array

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

**Parameters** `x` : array\_like

**chunks** : int, tuple

How to chunk the array. 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)).

-1 as a blocksize indicates the size of the corresponding dimension.

**name** : str, optional

The key name to use for the array. Defaults to a hash of `x`. Use `name=False` to generate a random name instead of hashing (fast)

**lock** : bool or Lock, optional

If `x` doesn't support concurrent reads then provide a lock here, or pass in `True` to have `dask.array` create one for you.

**asarray** : bool, optional

If `True` (default), then chunks will be converted to instances of `ndarray`. Set to `False` to pass passed chunks through unchanged.

**fancy** : bool, optional

If `x` doesn't support fancy indexing (e.g. indexing with lists or arrays) then set to `False`. Default is `True`.

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

`dask.array.store` (*sources, targets, lock=True, regions=None, compute=True, return\_stored=False, \*\*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.

**regions**: tuple of slices or iterable of tuple of slices

Each `region` tuple in `regions` should be such that `target[region].shape = source.shape` for the corresponding source and target in `sources` and `targets`, respectively.

**compute**: boolean, optional

If true compute immediately, return `dask.delayed.Delayed` otherwise

**return\_stored**: boolean, optional

Optionally return the stored result (default `False`).

**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.

This assumes that `k` is small. All results will be returned in a single chunk.

**Examples**

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

`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*, *allow\_unknown\_chunksizes=False*)

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)

**Parameters** *seq*: list of `dask.arrays`

**axis**: int

Dimension along which to align all of the arrays

**allow\_unknown\_chunksizes**: bool

Allow unknown chunksizes, such as come from converting from dask dataframes. Dask.array is unable to verify that chunks line up. If data comes from differently aligned sources then this can cause unexpected results.

**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*, *out=None*, *keepdims=False*)

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 to perform a logical AND 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.allclose` (*a*, *b*, *rtol*=1e-05, *atol*=1e-08, *equal\_nan*=False)

Returns True if 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*.

If either array contains one or more NaNs, False is returned. Infs are treated as equal if they are in the same place and of the same sign in both arrays.

**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.

New in version 1.10.0.

**Returns** `allclose` : bool

Returns True if the two arrays are equal within the given tolerance; False otherwise.

**See also:**

*isclose*, *all*, *any*

## Notes

If the following equation is element-wise True, then `allclose` returns True.

$$\text{absolute}(a - b) \leq (\text{atol} + \text{rtol} * \text{absolute}(b))$$

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

## Examples

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

```
>>> np.allclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
True
```

`dask.array.angle` (*x*, *deg=0*)

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 or 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*, *out=None*, *keepdims=False*)

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 to 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.apply_along_axis`** (*func1d*, *axis*, *arr*, *\*args*, *\*\*kwargs*)

Apply a function to 1-D slices along the given axis.

Execute *func1d(a, \*args)* where *func1d* operates on 1-D arrays and *a* is a 1-D slice of *arr* along *axis*.

**Parameters** **`func1d`** : function

This function should accept 1-D arrays. It is applied to 1-D slices of *arr* along the specified axis.

**`axis`** : integer

Axis along which *arr* is sliced.

**`arr`** : *ndarray*

Input array.

**`args`** : any

Additional arguments to *func1d*.

**kwargs: any**

Additional named arguments to *func1d*.

New in version 1.9.0.

**Returns** `apply_along_axis` : ndarray

The output array. The shape of *outarr* is identical to the shape of *arr*, except along the *axis* dimension, where the length of *outarr* is equal to the size of the return value of *func1d*. If *func1d* returns a scalar *outarr* will have one fewer dimensions than *arr*.

**See also:**

`apply_over_axes` Apply a function repeatedly over multiple axes.

## Examples

```
>>> def my_func(a):
...     """Average first and last element of a 1-D array"""
...     return (a[0] + a[-1]) * 0.5
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(my_func, 0, b)
array([ 4.,  5.,  6.])
>>> np.apply_along_axis(my_func, 1, b)
array([ 2.,  5.,  8.])
```

For a function that doesn't return a scalar, the number of dimensions in *outarr* is the same as *arr*.

```
>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
>>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
       [3, 4, 9],
       [2, 5, 6]])
```

`dask.array.apply_over_axes` (*func*, *a*, *axes*)

Apply a function repeatedly over multiple axes.

*func* is called as *res = func(a, axis)*, where *axis* is the first element of *axes*. The result *res* of the function call must have either the same dimensions as *a* or one less dimension. If *res* has one less dimension than *a*, a dimension is inserted before *axis*. The call to *func* is then repeated for each axis in *axes*, with *res* as the first argument.

**Parameters** `func` : function

This function must take two arguments, *func(a, axis)*.

**a** : array\_like

Input array.

**axes** : array\_like

Axes over which *func* is applied; the elements must be integers.

**Returns** `apply_over_axis` : ndarray

The output array. The number of dimensions is the same as *a*, but the shape can be different. This depends on whether *func* changes the shape of its output with respect to its input.

See also:

[`apply\_along\_axis`](#) Apply a function to 1-D slices of an array along the given axis.

## Notes

This function is equivalent to tuple axis arguments to reorderable ufuncs with `keepdims=True`. Tuple axis arguments to ufuncs have been available since version 1.7.0.

## Examples

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
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]])
```

Sum over axes 0 and 2. The result has same number of dimensions as the original array:

```
>>> np.apply_over_axes(np.sum, a, [0,2])
array([[[ 60],
        [ 92],
        [124]])])
```

Tuple axis arguments to ufuncs are equivalent:

```
>>> np.sum(a, axis=(0,2), keepdims=True)
array([[[ 60],
        [ 92],
        [124]])])
```

`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

**See also:**

`dask.array.linspace`

`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  $[0, \infty]$ .

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

For complex-valued input, *arccosh* is a complex analytical function that has a branch cut  $[-\infty, 1]$  and is continuous from above on it.

## References

[R103], [R104]

## 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  $[-\infty, -1]$  and  $[1, \infty]$  and is continuous from above on the former and from below on the latter.

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

## References

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

## Examples

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

`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

[R105], [R106]

## 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  $\pm\pi/2$ ). 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: [R107]

$x1$	$x2$	$\arctan2(x1,x2)$
+/- 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

[R107]

## 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

[R108], [R109]

## Examples

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

`dask.array.argmax` (*a*, *axis=None*, *out=None*)

Returns the 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.

**out** : array, optional

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

**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.argmaxin` (*a*, *axis=None*, *out=None*)

Returns the indices of the minimum 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.

**out** : array, optional

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

**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.argmin`, `argmax`

**amin** The minimum value along a given axis.

**unravel\_index** Convert a flat index into an index tuple.

## Notes

In case of multiple occurrences of the minimum 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.argmin(a)
0
>>> np.argmin(a, axis=0)
array([0, 0, 0])
>>> np.argmin(a, axis=1)
array([0, 0])
```

```

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

```

`dask.array.argmaxwhere(a)`

Find the indices of array elements that are non-zero, grouped by element.

**Parameters** `a` : array\_like

Input data.

**Returns** `index_array` : ndarray

Indices of elements that are non-zero. Indices are grouped by element.

**See also:**

*where, nonzero*

### Notes

`np.argmaxwhere(a)` is the same as `np.transpose(np.nonzero(a))`.

The output of `argwhere` is not suitable for indexing arrays. For this purpose use `where(a)` instead.

### Examples

```

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

```

`dask.array.around(a, decimals=0, out=None)`

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 [R110] and errors introduced when scaling by powers of ten.

**References**

[R110], [R111]

**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', 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' (default), then the returned array may be in any order (either C-, Fortran-contiguous, or even discontinuous), unless a copy is required, in which case it will be C-contiguous.

**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.**asanyarray**(*a*)

Convert the input to a dask array.

Subclasses of `np.ndarray` will be passed through as chunks unchanged.

**Parameters** *a* : array-like

Input data, in any form that can be converted to a dask array.

**Returns** *out* : dask array

Dask array interpretation of *a*.

### Examples

```
>>> import dask.array as da
>>> import numpy as np
>>> x = np.arange(3)
>>> da.asanyarray(x)
dask.array<array, shape=(3,), dtype=int64, chunksize=(3)>
```

```
>>> y = [[1, 2, 3], [4, 5, 6]]
>>> da.asanyarray(y)
dask.array<array, shape=(2, 3), dtype=int64, chunksize=(2, 3)>
```

dask.array.**asarray**(*a*)

Convert the input to a dask array.

**Parameters** *a* : array-like

Input data, in any form that can be converted to a dask array.

**Returns** *out* : dask array

Dask array interpretation of *a*.

### Examples

```
>>> import dask.array as da
>>> import numpy as np
>>> x = np.arange(3)
>>> da.asarray(x)
dask.array<array, shape=(3,), dtype=int64, chunksize=(3)>
```

```
>>> y = [[1, 2, 3], [4, 5, 6]]
>>> da.asarray(y)
dask.array<array, shape=(2, 3), dtype=int64, chunksize=(2, 3)>
```

`dask.array.atleast_1d(*arys)`

Convert inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters** `arys1, arys2, ...` : array\_like

One or more input arrays.

**Returns** `ret` : ndarray

An array, or sequence of arrays, each with `a.ndim >= 1`. Copies are made only if necessary.

**See also:**

`atleast_2d`, `atleast_3d`

### Examples

```
>>> np.atleast_1d(1.0)
array([ 1.])
```

```
>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> np.atleast_1d(x) is x
True
```

```
>>> np.atleast_1d(1, [3, 4])
[array([1]), array([3, 4])]
```

`dask.array.atleast_2d(*arys)`

View inputs as arrays with at least two dimensions.

**Parameters** `arys1, arys2, ...` : array\_like

One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

**Returns** `res, res2, ...` : ndarray

An array, or tuple of arrays, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

**See also:**

`atleast_1d`, `atleast_3d`

### Examples

```
>>> np.atleast_2d(3.0)
array([[ 3.]])
```

```
>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[ 0.,  1.,  2.]])
>>> np.atleast_2d(x).base is x
True
```

```
>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]
```

dask.array.**atleast\_3d**(\*args)

View inputs as arrays with at least three dimensions.

**Parameters** *args1, args2, ...* : array\_like

One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

**Returns** *res1, res2, ...* : ndarray

An array, or tuple of arrays, each with `a.ndim >= 3`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape  $(N,)$  becomes a view of shape  $(1, N, 1)$ , and a 2-D array of shape  $(M, N)$  becomes a view of shape  $(M, N, 1)$ .

**See also:**

[`atleast\_1d`](#), [`atleast\_2d`](#)

## Examples

```
>>> np.atleast_3d(3.0)
array([[[ 3.]])]
```

```
>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x
True
```

```
>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...     print(arr, arr.shape)
...
[[[1]
 [2]]] (1, 2, 1)
[[[1]
 [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)
```

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

A minimum number of bins for the output array.

New in version 1.6.0.

**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.block` (*arrays*, *allow\_unknown\_chunksizes=False*)

Assemble an nd-array from nested lists of blocks.

Blocks in the innermost lists are ‘concatenate’d along the last dimension (-1), then these are ‘concatenate’d along the second-last dimension (-2), and so on until the outermost list is reached

Blocks can be of any dimension, but will not be broadcasted using the normal rules. Instead, leading axes of size 1 are inserted, to make `block.ndim` the same for all blocks. This is primarily useful for working with scalars, and means that code like `block([v, 1])` is valid, where `v.ndim == 1`.

When the nested list is two levels deep, this allows block matrices to be constructed from their components.

**Parameters** `arrays` : nested list of array\_like or scalars (but not tuples)

If passed a single ndarray or scalar (a nested list of depth 0), this is returned unmodified (and not copied).

Elements shapes must match along the appropriate axes (without broadcasting), but leading 1s will be prepended to the shape as necessary to make the dimensions match.

**allow\_unknown\_chunksizes: bool**

Allow unknown chunksizes, such as come from converting from dask dataframes. Dask.array is unable to verify that chunks line up. If data comes from differently aligned sources then this can cause unexpected results.

**Returns** `block_array` : ndarray

The array assembled from the given blocks.

The dimensionality of the output is equal to the greatest of: \* the dimensionality of all the inputs \* the depth to which the input list is nested

**Raises** `ValueError`

- If list depths are mismatched - for instance, `[[a, b], c]` is illegal, and should be spelt `[[a, b], [c]]`
- If lists are empty - for instance, `[[a, b], []]`

**See also:**

`concatenate` Join a sequence of arrays together.

`stack` Stack arrays in sequence along a new dimension.

`hstack` Stack arrays in sequence horizontally (column wise).

`vstack` Stack arrays in sequence vertically (row wise).

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

`vsplit` Split array into a list of multiple sub-arrays vertically.

## Notes

When called with only scalars, `block` is equivalent to an ndarray call. So `block([[1, 2], [3, 4]])` is equivalent to `array([[1, 2], [3, 4]])`.

This function does not enforce that the blocks lie on a fixed grid. `block([[a, b], [c, d]])` is not restricted to arrays of the form:

```
AAAbb
AAAbb
cccDD
```

But is also allowed to produce, for some  $a$ ,  $b$ ,  $c$ ,  $d$ :

```
AAAbb
AAAbb
cDDDD
```

Since concatenation happens along the last axis first, *block* is `_not_` capable of producing the following directly:

```
AAAbb
cccbb
cccDD
```

Matlab's "square bracket stacking", `[A, B, ...; p, q, ...]`, is equivalent to `block([[A, B, ..
.], [p, q, ...]])`.

`dask.array.broadcast_to(x, shape, chunks=None)`

Broadcast an array to a new shape.

**Parameters** `x` : array\_like

The array to broadcast.

**shape** : tuple

The shape of the desired array.

**chunks** : tuple, optional

If provided, then the result will use these chunks instead of the same chunks as the source array. Setting chunks explicitly as part of `broadcast_to` is more efficient than rechunking afterwards. Chunks are only allowed to differ from the original shape along dimensions that are new on the result or have size 1 the input array.

**Returns** `broadcast` : dask array

**See also:**

`numpy.broadcast_to`

`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  $i \geq x$ . It is often denoted as  $\lceil x \rceil$ .

**Parameters** `x`: array\_like

Input data.

**Returns** `y`: ndarray or 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, out=None, mode='raise')`

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]*,  $i = 0, \dots, n-1$  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  $(j_0, j_1, \dots, j_m)$  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*, *out=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*, *allow\_unknown\_chunksizes=False*)

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)

**Parameters** *seq*: list of `dask.arrays`

**axis**: int

Dimension along which to align all of the arrays

**allow\_unknown\_chunksizes**: bool

Allow unknown chunksizes, such as come from converting from dask dataframes. Dask.array is unable to verify that chunks line up. If data comes from differently aligned sources then this can cause unexpected results.

**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*, *bias=<class 'numpy.\_NoValue'>*, *ddof=<class 'numpy.\_NoValue'>*)

Return Pearson product-moment correlation coefficients.

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

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

The values of *R* 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 *x* 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 *x*.

**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** : \_NoValue, optional

Has no effect, do not use.

Deprecated since version 1.10.0.

**ddof** : \_NoValue, optional

Has no effect, do not use.

Deprecated since version 1.10.0.

**Returns** *R* : ndarray

The correlation coefficient matrix of the variables.

**See also:**

[`cov`](#) Covariance matrix

## Notes

Due to floating point rounding the resulting array may not be Hermitian, the diagonal elements may not be 1, and the elements may not satisfy the inequality  $\text{abs}(a) \leq 1$ . The real and imaginary parts are clipped to the interval  $[-1, 1]$  in an attempt to improve on that situation but is not much help in the complex case.

This function accepts but discards arguments *bias* and *ddof*. This is for backwards compatibility with previous versions of this function. These arguments had no effect on the return values of the function and can be safely ignored in this and previous versions of numpy.

`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  $1/2 * (\text{np.exp}(x) + \text{np.exp}(-x))$  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.**count\_nonzero**(*a*)

Counts the number of non-zero values in the array *a*.

**Parameters** *a* : array\_like

The array for which to count non-zeros.

**Returns** *count* : int or array of int

Number of non-zero values in the array.

**See also:**

[\*nonzero\*](#) Return the coordinates of all the non-zero values.

## Examples

```
>>> np.count_nonzero(np.eye(4))
4
>>> np.count_nonzero([[0, 1, 7, 0, 0], [3, 0, 0, 2, 19]])
5
```

dask.array.**cov**(*m*, *y=None*, *rowvar=True*, *bias=False*, *ddof=None*, *fweights=None*, *aweights=None*)

Estimate a covariance matrix, given data and weights.

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

See the notes for an outline of the algorithm.

**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** : bool, optional

If *rowvar* is True (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** : bool, optional

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

**ddof** : int, optional

If not None the default value implied by *bias* is overridden. Note that *ddof*=1 will return the unbiased estimate, even if both *fweights* and *aweights* are specified, and *ddof*=0 will return the simple average. See the notes for the details. The default value is None.

New in version 1.5.

**fweights** : array\_like, int, optional

1-D array of integer frequency weights; the number of times each observation vector should be repeated.

New in version 1.10.

**aweights** : array\_like, optional

1-D array of observation vector weights. These relative weights are typically large for observations considered “important” and smaller for observations considered less “important”. If *ddof*=0 the array of weights can be used to assign probabilities to observation vectors.

New in version 1.10.

**Returns out** : ndarray

The covariance matrix of the variables.

**See also:**

[\*corrcoef\*](#) Normalized covariance matrix

## Notes

Assume that the observations are in the columns of the observation array *m* and let *f* = *fweights* and *a* = *aweights* for brevity. The steps to compute the weighted covariance are as follows:

```
>>> w = f * a
>>> v1 = np.sum(w)
>>> v2 = np.sum(w * a)
>>> m -= np.sum(m * w, axis=1, keepdims=True) / v1
>>> cov = np.dot(m * w, m.T) * v1 / (v1**2 - ddof * v2)
```

Note that when *a* == 1, the normalization factor  $v1 / (v1**2 - ddof * v2)$  goes over to  $1 / (np.sum(f) - ddof)$  as it should.

## 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(a, axis=None, dtype=None, out=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(a, axis=None, dtype=None, out=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 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*, *k*=0)

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 triangle 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.diff(a, n=1, axis=-1)`

Calculate the  $n$ -th discrete difference along given axis.

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

**Parameters** **a** : array\_like

Input array

**n** [int, optional] The number of times values are differenced.

**axis** [int, optional] The axis along which the difference is taken, default is the last axis.

**Returns** **diff** : ndarray

The  $n$ -th differences. The shape of the output is the same as *a* except along *axis* where the dimension is smaller by *n*.

## Examples

```
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.diff(x)
array([ 1,  2,  3, -7])
>>> np.diff(x, n=2)
array([ 1,  1, -10])
```

```
>>> x = np.array([[1, 3, 6, 10], [0, 5, 6, 8]])
>>> np.diff(x)
array([[2, 3, 4],
       [5, 1, 2]])
>>> np.diff(x, axis=0)
array([[ -1,  2,  0, -2]])
```

`dask.array.digitize(x, bins, right=False)`

Return the indices of the bins to which each value in input array belongs.

Each index  $i$  returned is such that  $\text{bins}[i-1] \leq x < \text{bins}[i]$  if  $\text{bins}$  is monotonically increasing, or  $\text{bins}[i-1] > x \geq \text{bins}[i]$  if  $\text{bins}$  is monotonically decreasing. If values in  $x$  are beyond the bounds of  $\text{bins}$ , 0 or  $\text{len}(\text{bins})$  is returned as appropriate. If  $\text{right}$  is `True`, then the right bin is closed so that the index  $i$  is such that  $\text{bins}[i-1] < x \leq \text{bins}[i]$  or  $\text{bins}[i-1] \geq x > \text{bins}[i]$  if  $\text{bins}$  is monotonically increasing or decreasing, respectively.

**Parameters**  $x$  : array\_like

Input array to be binned. Prior to Numpy 1.10.0, this array had to be 1-dimensional, but can now have any shape.

**bins** : array\_like

Array of bins. It has to be 1-dimensional and monotonic.

**right** : bool, optional

Indicating whether the intervals include the right or the left bin edge. Default behavior is (`right==False`) indicating that the interval does not include the right edge. The left bin end is open in this case, i.e.,  $\text{bins}[i-1] \leq x < \text{bins}[i]$  is the default behavior for monotonically increasing bins.

**Returns** **out** : ndarray of ints

Output array of indices, of same shape as  $x$ .

**Raises** **ValueError**

If  $\text{bins}$  is not monotonic.

**TypeError**

If the type of the input is complex.

**See also:**

[`bincount`](#), [`histogram`](#), [`unique`](#)

## Notes

If values in  $x$  are such that they fall outside the bin range, attempting to index  $\text{bins}$  with the indices that `digitize` returns will result in an `IndexError`.

New in version 1.10.0.

`np.digitize` is implemented in terms of `np.searchsorted`. This means that a binary search is used to bin the values, which scales much better for larger number of bins than the previous linear search. It also removes the requirement for the input array to be 1-dimensional.

## Examples

```
>>> x = np.array([0.2, 6.4, 3.0, 1.6])
>>> bins = np.array([0.0, 1.0, 2.5, 4.0, 10.0])
>>> inds = np.digitize(x, bins)
>>> inds
array([1, 4, 3, 2])
>>> for n in range(x.size):
```

```
... print(bins[inds[n]-1], "<=", x[n], "<", bins[inds[n]])
...
0.0 <= 0.2 < 1.0
4.0 <= 6.4 < 10.0
2.5 <= 3.0 < 4.0
1.0 <= 1.6 < 2.5
```

```
>>> x = np.array([1.2, 10.0, 12.4, 15.5, 20.])
>>> bins = np.array([0, 5, 10, 15, 20])
>>> np.digitize(x,bins,right=True)
array([1, 2, 3, 4, 4])
>>> np.digitize(x,bins,right=False)
array([1, 3, 3, 4, 5])
```

`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.

[`matmul`](#) '@' operator as method with out parameter.

## Examples

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

Neither argument is complex-conjugated:

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

For 2-D arrays it is 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:**

*stack* Join a sequence of arrays along a new axis.

*vstack* Stack along first axis.

*hstack* Stack along second axis.

*concatenate* Join a sequence of arrays along an existing axis.

*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.ediff1d` (*ary*, *to\_end=None*, *to\_begin=None*)  
The differences between consecutive elements of an array.

**Parameters** *ary* : array\_like

If necessary, will be flattened before the differences are taken.

**to\_end** : array\_like, optional

Number(s) to append at the end of the returned differences.

**to\_begin** : array\_like, optional

Number(s) to prepend at the beginning of the returned differences.

**Returns** `ediff1d` : ndarray

The differences. Loosely, this is `ary.flat[1:] - ary.flat[:-1]`.

**See also:**

`diff`, `gradient`

## Notes

When applied to masked arrays, this function drops the mask information if the *to\_begin* and/or *to\_end* parameters are used.

## Examples

```
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.ediff1d(x)
array([ 1,  2,  3, -7])
```

```
>>> np.ediff1d(x, to_begin=-99, to_end=np.array([88, 99]))
array([-99,  1,  2,  3, -7, 88, 99])
```

The returned array is always 1D.

```
>>> y = [[1, 2, 4], [1, 6, 24]]
>>> np.ediff1d(y)
array([ 1,  2, -3,  5, 18])
```

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

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 row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns** `out` : ndarray

Array of uninitialized (arbitrary) data of the given shape, dtype, and order. Object arrays will be initialized to `None`.

**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.empty_like(a, dtype=None, chunks=None)`

Return a new array with the same shape and type as a given array.

**Parameters** `a` : array\_like

The shape and data-type of `a` define these same attributes of the returned array.

**dtype** : data-type, optional

Overrides the data type of the result.

**chunks** : sequence of ints

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

**Returns out** : ndarray

Array of uninitialized (arbitrary) data with the same shape and type as *a*.

**See also:**

***ones\_like*** Return an array of ones with shape and type of input.

***zeros\_like*** Return an array of zeros with shape and type of input.

***empty*** Return a new uninitialized array.

***ones*** Return a new array setting values to one.

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

## Notes

This function does *not* initialize the returned array; to do that use *zeros\_like* or *ones\_like* instead. It may be marginally faster than the functions that do set the array values.

`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

[R112], [R113]

## 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=<class '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 or 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.flatnonzero(a)`

Return indices that are non-zero in the flattened version of a.

This is equivalent to `a.ravel().nonzero()[0]`.

**Parameters a** : ndarray

Input array.

**Returns res** : ndarray

Output array, containing the indices of the elements of `a.ravel()` that are non-zero.

**See also:**

**nonzero** Return the indices of the non-zero elements of the input array.

**ravel** Return a 1-D array containing the elements of the input array.

## Examples

```
>>> x = np.arange(-2, 3)
>>> x
array([-2, -1,  0,  1,  2])
>>> np.flatnonzero(x)
array([0, 1, 3, 4])
```

Use the indices of the non-zero elements as an index array to extract these elements:

```
>>> x.ravel()[np.flatnonzero(x)]
array([-2, -1,  1,  2])
```

`dask.array.flip(m, axis)`

Reverse element order along axis.

**Parameters axis** : int

Axis to reverse element order of.

**Returns reversed array** : ndarray

dask.array.**flipud**(*m*)

Flip array in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

**Parameters** *m* : array\_like

Input array.

**Returns** *out* : array\_like

A view of *m* with the rows reversed. Since a view is returned, this operation is  $\mathcal{O}(1)$ .

**See also:**

*fliplr* Flip array in the left/right direction.

**rot90** Rotate array counterclockwise.

## Notes

Equivalent to `A[::-1, ...]`. Does not require the array to be two-dimensional.

## Examples

```
>>> A = np.diag([1.0, 2, 3])
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
>>> np.flipud(A)
array([[ 0.,  0.,  3.],
       [ 0.,  2.,  0.],
       [ 1.,  0.,  0.]])
```

```
>>> A = np.random.randn(2,3,5)
>>> np.all(np.flipud(A)==A[::-1,...])
True
```

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

dask.array.**fliplr**(*m*)

Flip array in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

**Parameters** *m* : array\_like

Input array, must be at least 2-D.

**Returns** *f* : ndarray

A view of *m* with the columns reversed. Since a view is returned, this operation is  $\mathcal{O}(1)$ .

**See also:**

*flipud* Flip array in the up/down direction.

**rot90** Rotate array counterclockwise.

**Notes**

Equivalent to `A[:,::-1]`. Requires the array to be at least 2-D.

**Examples**

```
>>> A = np.diag([1.,2.,3.])
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
>>> np.fliplr(A)
array([[ 0.,  0.,  1.],
       [ 0.,  2.,  0.],
       [ 3.,  0.,  0.]])
```

```
>>> A = np.random.randn(2,3,5)
>>> np.all(np.fliplr(A)==A[:,::-1,...])
True
```

`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  $i \leq x$ . It is often denoted as  $\lfloor x \rfloor$ .

**Parameters** `x` : array\_like

Input data.

**Returns** `y` : ndarray or 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 or scalar

The maximum 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 or 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[, out1, out2])`

Decompose the elements of *x* into mantissa and twos exponent.

Returns (*mantissa*, *exponent*), where  $x = mantissa * 2^{**exponent}$ . The mantissa is lies in the open interval(-1, 1), while the twos exponent is a signed integer.

**Parameters** *x* : array\_like

Array of numbers to be decomposed.

**out1** : ndarray, optional

Output array for the mantissa. Must have the same shape as *x*.

**out2** : ndarray, optional

Output array for the exponent. Must have the same shape as *x*.

**Returns** (*mantissa*, *exponent*) : tuple of ndarrays, (float, int)

*mantissa* is a float array with values between -1 and 1. *exponent* is an int array which represents the exponent of 2.

**See also:**

[\*ldexp\*](#) Compute  $y = x1 * 2^{**x2}$ , the inverse of *frexp*.

## Notes

Complex dtypes are not supported, they will raise a `TypeError`.

## Examples

```
>>> x = np.arange(9)
>>> y1, y2 = np.frexp(x)
>>> y1
array([ 0.    ,  0.5   ,  0.5   ,  0.75  ,  0.5   ,  0.625,  0.75  ,  0.875,
        0.5   ])
>>> y2
array([0, 1, 2, 2, 3, 3, 3, 3, 4])
>>> y1 * 2**y2
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
```

`dask.array.fromfunction` (*function*, *shape*, *\*\*kwargs*)

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.frompyfunc` (*func*, *nin*, *nout*)

Takes an arbitrary Python function and returns a Numpy ufunc.

Can be used, for example, to add broadcasting to a built-in Python function (see Examples section).

**Parameters** *func* : Python function object

An arbitrary Python function.

**nin** : int

The number of input arguments.

**nout** : int

The number of objects returned by *func*.

**Returns** *out* : ufunc

Returns a Numpy universal function (ufunc) object.

## Notes

The returned ufunc always returns PyObject arrays.

## Examples

Use `frompyfunc` to add broadcasting to the Python function `oct`:

```
>>> oct_array = np.frompyfunc(oct, 1, 1)
>>> oct_array(np.array((10, 30, 100)))
array([012, 036, 0144], dtype=object)
>>> np.array((oct(10), oct(30), oct(100))) # for comparison
array(['012', '036', '0144'],
      dtype='<S4')

```

`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., `np.int8`. Default is `float`, but will change to `np.array(fill_value).dtype` in a future release.

**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.full_like` (*a*, *fill\_value*, *dtype=None*, *chunks=None*)

Return a full array with the same shape and type as a given array.

**Parameters a** : array\_like

The shape and data-type of *a* define these same attributes of the returned array.

**fill\_value** : scalar

Fill value.

**dtype** : data-type, optional

Overrides the data type of the result.

**chunks** : sequence of ints

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

**Returns out** : ndarray

Array of *fill\_value* with the same shape and type as *a*.

**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.

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

***ones*** Return a new array setting values to one.

**empty** Return a new uninitialized array.

**full** Fill a new array.

`dask.array.histogram` (*a*, *bins=None*, *range=None*, *normed=False*, *weights=None*, *density=None*)  
Blocked variant of `numpy.histogram`.

Follows the signature of `numpy.histogram` exactly with the following exceptions:

- Either an iterable specifying the `bins` or the number of bins and a `range` argument is required as computing `min` and `max` over blocked arrays is an expensive operation that must be performed explicitly.
- `weights` must be a `dask.array.Array` with the same block structure as `a`.

## Examples

Using number of bins and range:

```
>>> import dask.array as da
>>> import numpy as np
>>> x = da.from_array(np.arange(10000), chunks=10)
>>> h, bins = da.histogram(x, bins=10, range=[0, 10000])
>>> bins
array([    0.,   1000.,   2000.,   3000.,   4000.,   5000.,   6000.,
        7000.,   8000.,   9000.,  10000.])
>>> h.compute()
array([1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000])
```

Explicitly specifying the bins:

```
>>> h, bins = da.histogram(x, bins=np.array([0, 5000, 10000]))
>>> bins
array([    0,  5000, 10000])
>>> h.compute()
array([5000, 5000])
```

`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:**

**stack** Join a sequence of arrays along a new axis.

**vstack** Stack arrays in sequence vertically (row wise).

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

**concatenate** Join a sequence of arrays along an existing axis.

**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.indices` (*dimensions*, *dtype*=<class 'int'>, *chunks*=None)

Implements NumPy's `indices` for Dask Arrays.

Generates a grid of indices covering the dimensions provided.

The final array has the shape `(len(dimensions), *dimensions)`. The chunks are used to specify the chunking for axis 1 up to `len(dimensions)`. The 0th axis always has chunks of length 1.

**Parameters** *dimensions* : sequence of ints

The shape of the index grid.

**dtype** : dtype, optional

Type to use for the array. Default is `int`.

**chunks** : sequence of ints

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

**Returns** *grid* : dask array

`dask.array.insert` (*arr*, *obj*, *values*, *axis*=None)

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 along an existing axis.

**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` (*a*, *b*, *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.

$$\text{absolute}(a - b) \leq (\text{atol} + \text{rtol} * \text{absolute}(b))$$

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` (\*args, \*\*kwargs)

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 or 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` (*\*args, \*\*kwargs*)  
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  $num \% blocksize \neq 0$

**Returns** *samples* : dask array

**See also:**

[`dask.array.arange`](#)

`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

[R114], [R115]

## 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

[R116], [R117]

## 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  $[-\infty, -1]$  and is continuous from above on it. [`log1p`](#) handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

## References

[R118], [R119]

## 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  $[-\infty, 0]$  and is continuous from above on it. `log2` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

## Examples

```
>>> x = np.array([0, 1, 2, 2**4])
>>> np.log2(x)
array([-Inf,  0.,  1.,  4.]
```

```
>>> xi = np.array([0+1.j, 1, 2+0.j, 4.j])
>>> np.log2(xi)
array([ 0.+2.26618007j,  0.+0.j,  1.+0.j,  2.+2.26618007j])
```

`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)
3.50000000000000057e-50
```

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

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

Calculates  $\log_2(2^{x1} + 2^{x2})$ . This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

**Parameters** `x1, x2` : array\_like

Input values.

`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 or 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 or 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.matmul(a, b, out=None)`

Matrix product of two arrays.

The behavior depends on the arguments in the following way.

- If both arguments are 2-D they are multiplied like conventional matrices.
- If either argument is N-D,  $N > 2$ , it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.
- If the first argument is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions. After matrix multiplication the prepended 1 is removed.
- If the second argument is 1-D, it is promoted to a matrix by appending a 1 to its dimensions. After matrix multiplication the appended 1 is removed.

Multiplication by a scalar is not allowed, use `*` instead. Note that multiplying a stack of matrices with a vector will result in a stack of vectors, but `matmul` will not recognize it as such.

`matmul` differs from `dot` in two important ways.

- Multiplication by scalars is not allowed.

- Stacks of matrices are broadcast together as if the matrices were elements.

**Warning:** This function is preliminary and included in Numpy 1.10 for testing and documentation. Its semantics will not change, but the number and order of the optional arguments will.

New in version 1.10.0.

**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 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*.

If scalar value is passed.

**See also:**

`vdot` Complex-conjugating dot product.

`tensordot` Sum products over arbitrary axes.

`einsum` Einstein summation convention.

`dot` alternative matrix product with different broadcasting rules.

## Notes

The `matmul` function implements the semantics of the `@` operator introduced in Python 3.5 following PEP465.

## Examples

For 2-D arrays it is the matrix product:

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

For 2-D mixed with 1-D, the result is the usual.

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

Broadcasting is conventional for stacks of arrays

```
>>> a = np.arange(2*2*4).reshape((2,2,4))
>>> b = np.arange(2*2*4).reshape((2,4,2))
>>> np.matmul(a,b).shape
(2, 2, 2)
>>> np.matmul(a,b)[0,1,1]
98
>>> sum(a[0,1,:] * b[0,:,1])
98
```

Vector, vector returns the scalar inner product, but neither argument is complex-conjugated:

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

Scalar multiplication raises an error.

```
>>> np.matmul([1,2], 3)
Traceback (most recent call last):
...
ValueError: Scalar operands are not allowed, use '*' instead
```

`dask.array.max` (*a*, *axis=None*, *out=None*, *keepdims=False*)  
Return the maximum of an array or maximum along an axis.

**Parameters** *a* : array\_like

Input data.

**axis** : None or int or tuple of ints, optional

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

If this is a tuple of ints, the maximum is selected over multiple axes, instead of a single axis or all the axes as before.

**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 or 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*, *out=None*, *keepdims=False*)

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** : None or int or tuple of ints, optional

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

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

**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)
0.550000000074505806
```

`dask.array.meshgrid(*xi, **kwargs)`

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays *x1, x2, ..., xn*.

Changed in version 1.9: 1-D and 0-D cases are allowed.

**Parameters** `x1, x2, ..., xn` : array\_like

1-D arrays representing the coordinates of a grid.

**indexing** : {'xy', 'ij'}, optional

Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.

New in version 1.7.0.

**sparse** : bool, optional

If True a sparse grid is returned in order to conserve memory. Default is False.

New in version 1.7.0.

**copy** : bool, optional

If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.

New in version 1.7.0.

**Returns** `X1, X2, ..., XN` : ndarray

For vectors `x1, x2, ..., xn` with lengths `Ni=len(xi)`, return `(N1, N2, N3, ..., Nn)` shaped arrays if `indexing='ij'` or `(N2, N1, N3, ..., Nn)` shaped arrays if `indexing='xy'` with the elements of `xi` repeated to fill the matrix along the first dimension for `x1`, the second for `x2` and so on.

See also:

`index_tricks.meshgrid` Construct a multi-dimensional “meshgrid” using indexing notation.

`index_tricks.ogrid` Construct an open multi-dimensional “meshgrid” using indexing notation.

## Notes

This function supports both indexing conventions through the indexing keyword argument. Giving the string 'ij' returns a meshgrid with matrix indexing, while 'xy' returns a meshgrid with Cartesian indexing. In the 2-D case with inputs of length M and N, the outputs are of shape (N, M) for 'xy' indexing and (M, N) for 'ij' indexing. In the 3-D case with inputs of length M, N and P, outputs are of shape (N, M, P) for 'xy' indexing and (M, N, P) for 'ij' indexing. The difference is illustrated by the following code snippet:

```
xv, yv = meshgrid(x, y, sparse=False, indexing='ij')
for i in range(nx):
    for j in range(ny):
        # treat xv[i,j], yv[i,j]

xv, yv = meshgrid(x, y, sparse=False, indexing='xy')
for i in range(nx):
    for j in range(ny):
        # treat xv[j,i], yv[j,i]
```

In the 1-D and 0-D case, the indexing and sparse keywords have no effect.

## Examples

```

>>> nx, ny = (3, 2)
>>> x = np.linspace(0, 1, nx)
>>> y = np.linspace(0, 1, ny)
>>> xv, yv = meshgrid(x, y)
>>> xv
array([[ 0. ,  0.5,  1. ],
       [ 0. ,  0.5,  1.]])
>>> yv
array([[ 0.,  0.,  0.],
       [ 1.,  1.,  1.]])
>>> xv, yv = meshgrid(x, y, sparse=True) # make sparse output arrays
>>> xv
array([[ 0. ,  0.5,  1. ]])
>>> yv
array([[ 0.],
       [ 1.]])

```

`meshgrid` is very useful to evaluate functions on a grid.

```

>>> x = np.arange(-5, 5, 0.1)
>>> y = np.arange(-5, 5, 0.1)
>>> xx, yy = meshgrid(x, y, sparse=True)
>>> z = np.sin(xx**2 + yy**2) / (xx**2 + yy**2)
>>> h = plt.contourf(x, y, z)

```

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

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

**Parameters** `a` : array\_like

Input data.

**axis** : None or int or tuple of ints, optional

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

If this is a tuple of ints, the minimum is selected over multiple axes, instead of a single axis or all the axes as before.

**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 or 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[, out1, out2])`

Return the fractional and integral parts of an array, element-wise.

The fractional and integral parts are negative if the given number is negative.

**Parameters** `x` : array\_like

Input array.

**Returns** `y1` : ndarray

Fractional part of `x`.

`y2` : ndarray

Integral part of `x`.

## Notes

For integer input the return values are floats.

## Examples

```
>>> np.modf([0, 3.5])
(array([ 0. ,  0.5]), array([ 0.,  3.]))
>>> np.modf(-0.5)
(-0.5, -0)
```

`dask.array.moment` (*a*, *order*, *axis=None*, *dtype=None*, *keepdims=False*, *ddof=0*, *split\_every=None*, *out=None*)

`dask.array.nanargmax` (*x*, *axis*, *\*\*kwargs*)

`dask.array.nanargmin` (*x*, *axis*, *\*\*kwargs*)

`dask.array.nancumprod` (*a*, *axis=None*, *dtype=None*, *out=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` (*a*, *axis=None*, *dtype=None*, *out=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*, *out=None*, *keepdims=False*)

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*, *out=None*, *keepdims=False*)

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*, *out=None*, *keepdims=False*)

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*, *out=None*, *keepdims=0*)

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*, *out=None*, *ddof=0*, *keepdims=False*)

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  $N - \text{ddof}$ , 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  $\geq$  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:  $\text{std} = \sqrt{\text{mean}(\text{abs}(x - x.\text{mean}())**2)}$ .

The average squared deviation is normally calculated as  $x.\text{sum}() / N$ , where  $N = \text{len}(x)$ . If, however, *ddof* is specified, the divisor  $N - \text{ddof}$  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, out=None, keepdims=0)`

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

In Numpy versions  $\leq 1.8$  Nan is returned for slices that are all-NaN or empty. In later versions zero is 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*, *out=None*, *ddof=0*, *keepdims=False*)

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  $N - \text{ddof}$ , 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  $\geq$  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.nonzero(a)`

Return the indices of the elements that are non-zero.

Returns a tuple of arrays, one for each dimension of *a*, containing the indices of the non-zero elements in that dimension. The values in *a* are always tested and returned in row-major, C-style order. The corresponding non-zero values can be obtained with:

```
a[nonzero(a)]
```

To group the indices by element, rather than dimension, use:

```
transpose(nonzero(a))
```

The result of this is always a 2-D array, with a row for each non-zero element.

**Parameters** *a* : array\_like

Input array.

**Returns** *tuple\_of\_arrays* : tuple

Indices of elements that are non-zero.

**See also:**

[\*flatnonzero\*](#) Return indices that are non-zero in the flattened version of the input array.

`ndarray.nonzero` Equivalent ndarray method.

[\*count\\_nonzero\*](#) Counts the number of non-zero elements in the input array.

## Examples

```
>>> x = np.eye(3)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))
```

```
>>> x[np.nonzero(x)]
array([ 1.,  1.,  1.])
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `np.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the boolean array can also be called.

```
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

`dask.array.notnull` (*values*)  
pandas.notnull for dask arrays

`dask.array.ones` (*\*args, \*\*kwargs*)  
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.ones_like` (*a*, *dtype=None*, *chunks=None*)

Return an array of ones with the same shape and type as a given array.

**Parameters** *a* : array\_like

The shape and data-type of *a* define these same attributes of the returned array.

**dtype** : data-type, optional

Overrides the data type of the result.

**chunks** : sequence of ints

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

**Returns** *out* : ndarray

Array of ones with the same shape and type as *a*.

**See also:**

[`zeros\_like`](#) Return an array of zeros with shape and type of input.

[`empty\_like`](#) Return an empty 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.

`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*, *out=None*, *keepdims=False*)

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`, will calculate the product of all the elements in the input array. If `axis` is negative it counts from the last to the first axis.

New in version 1.7.0.

If `axis` is a tuple of ints, a product is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

**dtype** : dtype, optional

The type of the returned array, as well as of the accumulator in which the elements are multiplied. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform

integer is used while if  $a$  is unsigned then an unsigned integer of the same precision as the platform integer is used.

**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 input array.

**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
```

The product of an empty array is the neutral element 1:

```
>>> np.prod([])
1.0
```

## 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.ptp` (*a*, *axis=None*, *out=None*)

Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

**Parameters** *a* : array\_like

Input values.

**axis** : int, optional

Axis along which to find the peaks. By default, flatten the array.

**out** : array\_like

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 output values will be cast if necessary.

**Returns** *ptp* : ndarray

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

## Examples

```
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])
```

```
>>> np.ptp(x, axis=0)
array([2, 2])
```

```
>>> np.ptp(x, axis=1)
array([1, 1])
```

`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(a, order='C')`

Return a contiguous flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

**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 row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style 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** `y` : array\_like

If `a` is a matrix, `y` is a 1-D ndarray, otherwise `y` is an array of the same subtype as `a`. The shape of the returned array is `(a.size,)`. Matrices are special cased for backward compatibility.

**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.

`ndarray.reshape` Change the shape of an array without changing its data.

## Notes

In row-major, C-style 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 column-major, Fortran-style index ordering.

When a view is desired in as many cases as possible, `arr.reshape(-1)` may be preferable.

## 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`, `threshold=4`, `block_size_limit=100000000.0`)

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**: tuple

The new block dimensions to create

#### **threshold**: int

The graph growth factor under which we don't bother introducing an intermediate step

#### **block\_size\_limit**: int

The maximum block size (in bytes) we want to produce during an intermediate step

`dask.array.repeat(a, repeats, axis=None)`

Repeat elements of an array.

### Parameters **a**: array\_like

Input array.

#### **repeats**: int or array of ints

The number of repetitions for each element. *repeats* is broadcasted to fit the shape of the given axis.

#### **axis**: int, optional

The axis along which to repeat values. By default, use the flattened input array, and return a flat output array.

### Returns **repeated\_array**: ndarray

Output array which has the same shape as *a*, except along the given axis.

### See also:

[`tile`](#) Tile an array.

### Examples

```
>>> x = np.array([[1,2],[3,4]])
>>> np.repeat(x, 2)
array([1, 1, 2, 2, 3, 3, 4, 4])
>>> np.repeat(x, 3, axis=1)
array([[1, 1, 1, 2, 2, 2],
```

```
[3, 3, 3, 4, 4, 4]])
>>> np.repeat(x, [1, 2], axis=0)
array([[1, 2],
       [3, 4],
       [3, 4]])
```

`dask.array.reshape(x, shape)`

Reshape array to new shape

This is a parallelized version of the `np.reshape` function with the following limitations:

1. It assumes that the array is stored in C-order
2. It only allows for reshaping that collapse or merge dimensions like  $(1, 2, 3, 4) \rightarrow (1, 6, 4)$  or  $(64,) \rightarrow (4, 4, 4)$

When communication is necessary this algorithm depends on the logic within `rechunk`. It endeavors to keep chunk sizes roughly the same when possible.

**See also:**

`dask.array.rechunk`, `numpy.reshape`

`dask.array.result_type(*arrays_and_dtypes)`

Returns the type that results from applying the NumPy type promotion rules to the arguments.

Type promotion in NumPy works similarly to the rules in languages like C++, with some slight differences. When both scalars and arrays are used, the array's type takes precedence and the actual value of the scalar is taken into account.

For example, calculating  $3*a$ , where `a` is an array of 32-bit floats, intuitively should result in a 32-bit float output. If the `3` is a 32-bit integer, the NumPy rules indicate it can't convert losslessly into a 32-bit float, so a 64-bit float should be the result type. By examining the value of the constant, '3', we see that it fits in an 8-bit integer, which can be cast losslessly into the 32-bit float.

**Parameters** `arrays_and_dtypes` : list of arrays and dtypes

The operands of some operation whose result type is needed.

**Returns** `out` : dtype

The result type.

**See also:**

`dtype`, `promote_types`, `min_scalar_type`, `can_cast`

## Notes

New in version 1.6.0.

The specific algorithm used is as follows.

Categories are determined by first checking which of boolean, integer (int/uint), or floating point (float/complex) the maximum kind of all the arrays and the scalars are.

If there are only scalars or the maximum category of the scalars is higher than the maximum category of the arrays, the data types are combined with `promote_types()` to produce the return value.

Otherwise, `min_scalar_type` is called on each array, and the resulting data types are all combined with `promote_types()` to produce the return value.

The set of int values is not a subset of the uint values for types with the same number of bits, something not reflected in `min_scalar_type()`, but handled as a special case in `result_type`.

### Examples

```
>>> np.result_type(3, np.arange(7, dtype='i1'))
dtype('int8')
```

```
>>> np.result_type('i4', 'c8')
dtype('complex128')
```

```
>>> np.result_type(3.0, -2)
dtype('float64')
```

`dask.array.rint` (`x`[, `out` ])

Round elements of the array to the nearest integer.

**Parameters** `x` : array\_like

Input array.

**Returns** `out` : ndarray or 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.roll` (`a`, `shift`, `axis=None`)

Roll array elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

**Parameters** `a` : array\_like

Input array.

**shift** : int

The number of places by which elements are shifted.

**axis** : int, optional

The axis along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

**Returns** `res` : ndarray

Output array, with the same shape as `a`.

**See also:**

`rollaxis` Roll the specified axis backwards, until it lies in a given position.

## Examples

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

```
>>> x2 = np.reshape(x, (2,5))
>>> x2
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> np.roll(x2, 1)
array([[9, 0, 1, 2, 3],
       [4, 5, 6, 7, 8]])
>>> np.roll(x2, 1, axis=0)
array([[5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4]])
>>> np.roll(x2, 1, axis=1)
array([[4, 0, 1, 2, 3],
       [9, 5, 6, 7, 8]])
```

dask.array.**round**(*a*, *decimals*=0, *out*=None)  
Round an array to the given number of decimals.

Refer to *around* for full documentation.

**See also:**

*around* equivalent function

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  $x > 0$ . nan is returned for nan inputs.

For complex inputs, the *sign* function returns  $\text{sign}(x.\text{real}) + 0j$  if  $x.\text{real} \neq 0$  else  $\text{sign}(x.\text{imag}) + 0j$ .

complex(nan, 0) is returned for complex nan inputs.

**Parameters** *x*: array\_like

Input values.

**Returns** *y*: ndarray

The sign of *x*.

## Notes

There is more than one definition of sign in common use for complex numbers. The definition used here is equivalent to  $x/\sqrt{x * \bar{x}}$  which is different from a common alternative,  $x/|x|$ .

## Examples

```
>>> np.sign([-5., 4.5])
array([-1.,  1.])
>>> np.sign(0)
0
>>> np.sign(5-2j)
(1+0j)
```

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 ( $2\pi$  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  $\pi$ . Sines of angles between  $\pi$  and  $2\pi$  are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

### Examples

Print sine of one angle:

```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.          ,  0.5          ,  0.70710678,  0.8660254 ,  1.          ])
```

Plot the sine function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

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 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*, *out=None*, *ddof=0*, *keepdims=False*)

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** : None or int or tuple of ints, optional

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

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

**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  $N - \text{ddof}$ , 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)
0.45000005
```

Computing the standard deviation in `float64` is more accurate:

```
>>> np.std(a, dtype=np.float64)
0.44999999925494177
```

`dask.array.sum(a, axis=None, dtype=None, out=None, keepdims=False)`

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`, will sum all of the elements of the input array. If `axis` is negative it counts from the last to the first axis.

New in version 1.7.0.

If `axis` is a tuple of ints, a sum is performed on all of the axes specified in the tuple 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. The dtype of `a` is used by default unless `a` has an integer dtype of less precision than the default platform integer. In that case, if `a` is signed then the platform integer is used while if `a` is unsigned then an unsigned integer of the same precision as the platform integer is used.

**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 input array.

**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.

The sum of an empty array is the neutral element 0:

```
>>> np.sum([])
0.0
```

## 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=None, out=None, mode='raise')`

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:**

`compress` Take elements using a boolean mask

`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  $\text{np.sinh}(x) / \text{np.cosh}(x)$  or  $-1j * \text{np.tan}(1j*x)$ .

**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

[R120], [R121]

## 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(a, b, axes=2)`

Compute tensor dot product along specified axes for arrays  $\geq$  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)  $\geq$  1`

Tensors to “dot”.

`axes` : int or (2,) `array_like`

- integer\_like If an int  $N$ , sum over the last  $N$  axes of  $a$  and the first  $N$  axes of  $b$  in order. The sizes of the corresponding axes must match.
- (2,) array\_like Or, a list of axes to be summed over, first sequence applying to  $a$ , second to  $b$ . Both elements `array_like` must be of the same length.

**See also:**

`dot`, `einsum`

## Notes

**Three common use cases are:** `axes = 0` : tensor product `aotimes b` `axes = 1` : tensor dot product `acdot b` `axes = 2` : (default) tensor double contraction `a:b`

When `axes` is integer\_like, the sequence for evaluation will be: first the  $-N$ th axis in  $a$  and 0th axis in  $b$ , and the  $-1$ th axis in  $a$  and  $N$ th axis in  $b$  last.

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 for double-contraction
array([abbccddddd, aaaaabbbbbccccccddddd], dtype=object)
```

```
>>> np.tensordot(a, A, 1)
array([[[acc, bdd],
        [aaacccc, bbbddddd]],
       [[aaaaaccccc, bbbbbbddddd],
        [aaaaaaaccccc, bbbbbbddddd]]], dtype=object)
```

```
>>> np.tensordot(a, A, 0) # tensor product (result too long to incl.)
array([[[[a, b],
         [c, d]],
        ...
```

```
>>> np.tensordot(a, A, (0, 1))
array([[abbbbb, cddddd],
       [aabbbbb, ccddddd],
       [aaabbbbb, cccddddd],
       [aaaabbbbb, ccccddddd]], dtype=object)
```

```
>>> np.tensordot(a, A, (2, 1))
array([[abb, cdd],
       [aaabbbb, cccddd],
       [aaaaabbbbb, ccccddddd],
       [aaaaaabbbbb, cccccddddd]]], dtype=object)
```

```
>>> np.tensordot(a, A, ((0, 1), (0, 1)))
array([abbcccccddddd, aabbbbccccccddddd], dtype=object)
```

```
>>> np.tensordot(a, A, ((2, 1), (1, 0)))
array([accbbddd, aaaaacccccbbbbbddddd], dtype=object)
```

`dask.array.tile(A, reps)`

Construct an array by repeating A the number of times given by reps.

If *reps* has length *d*, the result will have dimension of  $\max(d, A.\text{ndim})$ .

If  $A.\text{ndim} < d$ , *A* is promoted to be *d*-dimensional by prepending new axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication, or shape (1, 1, 3) for 3-D replication. If this is not the desired behavior, promote *A* to *d*-dimensions manually before calling this function.

If  $A.\text{ndim} > d$ , *reps* is promoted to *A.ndim* by pre-pending 1's to it. Thus for an *A* of shape (2, 3, 4, 5), a *reps* of (2, 2) is treated as (1, 1, 2, 2).

Note : Although `tile` may be used for broadcasting, it is strongly recommended to use numpy's broadcasting operations and functions.

**Parameters** *A* : array\_like

The input array.

**reps** : array\_like

The number of repetitions of *A* along each axis.

**Returns** *c* : ndarray

The tiled output array.

**See also:**

[\*repeat\*](#) Repeat elements of an array.

[\*broadcast\\_to\*](#) Broadcast an array to a new shape

## Examples

```
>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> np.tile(a, (2, 1, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
```

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

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

`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.

This assumes that  $k$  is small. All results will be returned in a single chunk.

### Examples

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

`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:**

`moveaxis`, `argsort`

### Notes

Use `transpose(a, argsort(axes))` to invert the transposition of tensors when using the `axes` keyword argument.

Transposing a 1-D array returns an unchanged view of the original array.

### 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 or 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` (*ar*, *return\_index=False*, *return\_inverse=False*, *return\_counts=False*)

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements: the indices of the input array that give the unique values, the indices of the unique array that reconstruct the input array, and the number of times each unique value comes up in 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*.

**return\_counts** : bool, optional

If True, also return the number of times each unique value comes up in *ar*.

New in version 1.9.0.

**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.

**unique\_counts** : ndarray, optional

The number of times each of the unique values comes up in the original array. Only provided if *return\_counts* is True.

New in version 1.9.0.

**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*, *out=None*, *ddof=0*, *keepdims=False*)

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** : None or int or tuple of ints, optional

Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

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

**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  $N - \text{ddof}$ , 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.,  $\text{var} = \text{mean}(\text{abs}(x - \text{mean}(x)) ** 2)$ .

The mean is normally calculated as  $x.\text{sum}() / N$ , where  $N = \text{len}(x)$ . If, however, *ddof* is specified, the divisor  $N - \text{ddof}$  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)
0.20250003
```

Computing the variance in *float64* is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932944759
>>> ((1-0.55)**2 + (0.1-0.55)**2) / 2
0.2025
```

`dask.array.vdot(a, b)`

Return the dot product of two vectors.

The `vdot(a, b)` function handles complex numbers differently than `dot(a, b)`. If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product.

Note that `vdot` handles multidimensional arrays differently than `dot`: it does *not* perform a matrix product, but flattens input arguments to 1-D vectors first. Consequently, it should only be used for vectors.

**Parameters** `a` : array\_like

If `a` is complex the complex conjugate is taken before calculation of the dot product.

`b` : array\_like

Second argument to the dot product.

**Returns** `output` : ndarray

Dot product of `a` and `b`. Can be an int, float, or complex depending on the types of `a` and `b`.

**See also:**

`dot` Return the dot product without using the complex conjugate of the first argument.

### Examples

```
>>> a = np.array([1+2j, 3+4j])
>>> b = np.array([5+6j, 7+8j])
>>> np.vdot(a, b)
(70-8j)
>>> np.vdot(b, a)
(70+8j)
```

Note that higher-dimensional arrays are flattened!

```
>>> a = np.array([[1, 4], [5, 6]])
>>> b = np.array([[4, 1], [2, 2]])
>>> np.vdot(a, b)
30
>>> np.vdot(b, a)
30
>>> 1*4 + 4*1 + 5*2 + 6*2
30
```

`dask.array.vnorm(a, ord=None, axis=None, dtype=None, keepdims=False, split_every=None, out=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:**

**stack** Join a sequence of arrays along a new axis.

**hstack** Stack arrays in sequence horizontally (column wise).

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

**concatenate** Join a sequence of arrays along an existing axis.

**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(*args, **kwargs)`

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=np.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.zeros_like(a, dtype=None, chunks=None)`

Return an array of zeros with the same shape and type as a given array.

**Parameters a** : array\_like

The shape and data-type of *a* define these same attributes of the returned array.

**dtype** : data-type, optional

Overrides the data type of the result.

**chunks** : sequence of ints

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

**Returns out** : ndarray

Array of zeros with the same shape and type as  $a$ .

**See also:**

**`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.

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

**`ones`** Return a new array setting values to one.

**`empty`** Return a new uninitialized array.

`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  $ax = b$  by computing a vector  $x$  that minimizes the Euclidean 2-norm  $\|b - ax\|^2$ . 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.norm(x, ord=None, axis=None, keepdims=False)`  
Matrix or vector norm.

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

**Parameters** **x** : array\_like

Input array. If *axis* is None, *x* must be 1-D or 2-D.

**ord** : {non-zero int, inf, -inf, 'fro', 'nuc'}, optional

Order of the norm (see table under Notes). *inf* means numpy's *inf* object.

**axis** : {int, 2-tuple of ints, None}, optional

If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned.

**keepdims** : bool, optional

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

New in version 1.10.0.

**Returns** **n** : float or ndarray

Norm of the matrix or vector(s).

## Notes

For values of `ord`  $\leq 0$ , the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for various numerical purposes.

The following norms can be calculated:

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	-
'nuc'	nuclear norm	-
inf	$\max(\text{sum}(\text{abs}(x), \text{axis}=1))$	$\max(\text{abs}(x))$
-inf	$\min(\text{sum}(\text{abs}(x), \text{axis}=1))$	$\min(\text{abs}(x))$
0	-	$\text{sum}(x \neq 0)$
1	$\max(\text{sum}(\text{abs}(x), \text{axis}=0))$	as below
-1	$\min(\text{sum}(\text{abs}(x), \text{axis}=0))$	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	-	$\text{sum}(\text{abs}(x)**\text{ord})*(1./\text{ord})$

The Frobenius norm is given by [R122]:

$$\|A\|_F = [\sum_{i,j} \text{abs}(a_{i,j})^2]^{1/2}$$

The nuclear norm is the sum of the singular values.

## References

[R122]

## Examples

```
>>> from numpy import linalg as LA
>>> a = np.arange(9) - 4
>>> a
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> b = a.reshape((3, 3))
>>> b
array([[ -4,  -3,  -2],
       [ -1,   0,   1],
       [  2,   3,   4]])
```

```
>>> LA.norm(a)
7.745966692414834
>>> LA.norm(b)
7.745966692414834
>>> LA.norm(b, 'fro')
7.745966692414834
>>> LA.norm(a, np.inf)
4.0
>>> LA.norm(b, np.inf)
9.0
>>> LA.norm(a, -np.inf)
0.0
>>> LA.norm(b, -np.inf)
2.0
```

```
>>> LA.norm(a, 1)
20.0
>>> LA.norm(b, 1)
```

```

7.0
>>> LA.norm(a, -1)
-4.6566128774142013e-010
>>> LA.norm(b, -1)
6.0
>>> LA.norm(a, 2)
7.745966692414834
>>> LA.norm(b, 2)
7.3484692283495345

```

```

>>> LA.norm(a, -2)
nan
>>> LA.norm(b, -2)
1.8570331885190563e-016
>>> LA.norm(a, 3)
5.8480354764257312
>>> LA.norm(a, -3)
nan

```

Using the *axis* argument to compute vector norms:

```

>>> c = np.array([[ 1, 2, 3],
...               [-1, 1, 4]])
>>> LA.norm(c, axis=0)
array([ 1.41421356,  2.23606798,  5.          ])
>>> LA.norm(c, axis=1)
array([ 3.74165739,  4.24264069])
>>> LA.norm(c, ord=1, axis=1)
array([ 6.,  6.])

```

Using the *axis* argument to compute matrix norms:

```

>>> m = np.arange(8).reshape(2,2,2)
>>> LA.norm(m, axis=(1,2))
array([ 3.74165739, 11.22497216])
>>> LA.norm(m[0, :, :]), LA.norm(m[1, :, :])
(3.7416573867739413, 11.224972160321824)

```

`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:** intNumber of power iterations, useful when the singular values decay slowly. Error decreases exponentially as `n_power_iter` increases. In practice, set `n_power_iter`  $\leq$  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.

If blocks are of size  $(n, k)$  then this algorithm has memory use that scales as  $n^{**2} * k * nthreads$ .

**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.ma.filled` (*a*, *fill\_value=None*)

Return input as an array with masked data replaced by a fill value.

If *a* is not a *MaskedArray*, *a* itself is returned. If *a* is a *MaskedArray* and *fill\_value* is *None*, *fill\_value* is set to *a.fill\_value*.

**Parameters a :** MaskedArray or array\_like

An input object.

**fill\_value** : scalar, optional

Filling value. Default is None.

**Returns a** : ndarray

The filled array.

**See also:**

compressed

## Examples

```
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
...                                                [1, 0, 0],
...                                                [0, 0, 0]])
>>> x.filled()
array([[999999,      1,      2],
       [999999,      4,      5],
       [      6,      7,      8]])
```

`dask.array.ma.fix_invalid(a, mask=False, copy=True, fill_value=None)`

Return input with invalid data masked and replaced by a fill value.

Invalid data means values of *nan*, *inf*, etc.

**Parameters a** : array\_like

Input array, a (subclass of) ndarray.

**mask** : sequence, optional

Mask. Must be convertible to an array of booleans with the same shape as *data*. True indicates a masked (i.e. invalid) data.

**copy** : bool, optional

Whether to use a copy of *a* (True) or to fix *a* in place (False). Default is True.

**fill\_value** : scalar, optional

Value used for fixing invalid data. Default is None, in which case the `a.fill_value` is used.

**Returns b** : MaskedArray

The input array with invalid entries fixed.

## Notes

A copy is performed by default.

## Examples

```
>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data = [-- -1.0 nan inf],
             mask = [ True False False False],
```

```

    fill_value = 1e+20)
>>> np.ma.fix_invalid(x)
masked_array(data = [-- -1.0 -- --],
             mask = [ True False  True  True],
             fill_value = 1e+20)

```

```

>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([ 1.00000000e+00, -1.00000000e+00,  1.00000000e+20,
        1.00000000e+20])
>>> x.data
array([ 1., -1., NaN, Inf])

```

dask.array.ma.**getdata** (*a*, *subok=True*)

Return the data of a masked array as an ndarray.

Return the data of *a* (if any) as an ndarray if *a* is a MaskedArray, else return *a* as a ndarray or subclass (depending on *subok*) if not.

**Parameters** *a* : array\_like

Input MaskedArray, alternatively a ndarray or a subclass thereof.

**subok** : bool

Whether to force the output to be a *pure* ndarray (False) or to return a subclass of ndarray if appropriate (True, default).

**See also:**

**getmask** Return the mask of a masked array, or nomask.

**getmaskarray** Return the mask of a masked array, or full array of False.

## Examples

```

>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
  [[1 --]
   [3 4]],
             mask =
  [[False  True]
   [False False]],
             fill_value=999999)
>>> ma.getdata(a)
array([[1, 2],
       [3, 4]])

```

Equivalently use the MaskedArray *data* attribute.

```

>>> a.data
array([[1, 2],
       [3, 4]])

```

dask.array.ma.**getmaskarray** (*arr*)

Return the mask of a masked array, or full boolean array of False.

Return the mask of *arr* as an ndarray if *arr* is a *MaskedArray* and the mask is not *nomask*, else return a full boolean array of False of the same shape as *arr*.

**Parameters** *arr* : array\_like

Input *MaskedArray* for which the mask is required.

**See also:**

**getmask** Return the mask of a masked array, or nomask.

**getdata** Return the data of a masked array as an ndarray.

## Examples

```
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
  [[1 --]
   [3 4]],
             mask =
  [[False True]
   [False False]],
             fill_value=999999)
>>> ma.getmaskarray(a)
array([[False,  True],
       [False, False]], dtype=bool)
```

Result when mask == nomask

```
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
  [[1 2]
   [3 4]],
             mask =
  False,
             fill_value=999999)
>>> >ma.getmaskarray(b)
array([[False, False],
       [False, False]], dtype=bool)
```

`dask.array.ma.masked_array` (*data=None, mask=False, dtype=None, copy=False, subok=True, ndmin=0, fill\_value=None, keep\_mask=True, hard\_mask=None, shrink=True, order=None, \*\*options*)

An array class with possibly masked values.

Masked values of True exclude the corresponding element from any computation.

Construction:

```
x = MaskedArray(data, mask=nomask, dtype=None, copy=False, subok=True,
                ndmin=0, fill_value=None, keep_mask=True, hard_mask=None,
                shrink=True, order=None)
```

**Parameters** *data* : array\_like

Input data.

**mask** : sequence, optional

Mask. Must be convertible to an array of booleans with the same shape as *data*. True indicates a masked (i.e. invalid) data.

**dtype** : dtype, optional

Data type of the output. If *dtype* is None, the type of the data argument (`data.dtype`) is used. If *dtype* is not None and different from `data.dtype`, a copy is performed.

**copy** : bool, optional

Whether to copy the input data (True), or to use a reference instead. Default is False.

**subok** : bool, optional

Whether to return a subclass of *MaskedArray* if possible (True) or a plain *MaskedArray*. Default is True.

**ndmin** : int, optional

Minimum number of dimensions. Default is 0.

**fill\_value** : scalar, optional

Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.

**keep\_mask** : bool, optional

Whether to combine *mask* with the mask of the input data, if any (True), or to use only *mask* for the output (False). Default is True.

**hard\_mask** : bool, optional

Whether to use a hard mask or not. With a hard mask, masked values cannot be unmasked. Default is False.

**shrink** : bool, optional

Whether to force compression of an empty mask. Default is True.

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

Specify the order of the array. If order is 'C', 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' (default), then the returned array may be in any order (either C-, Fortran-contiguous, or even discontinuous), unless a copy is required, in which case it will be C-contiguous.

`dask.array.ma.masked_equal(x, value, copy=True)`

Mask an array where equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x == value)`. For floating point arrays, consider using `masked_values(x, value)`.

**See also:**

[`masked\_where`](#) Mask where a condition is met.

[`masked\_values`](#) Mask using floating point equality.

## Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 -- 3],
             mask = [False False  True False],
             fill_value=999999)
```

`dask.array.ma.masked_greater` (*x*, *value*, *copy=True*)

Mask an array where greater than a given value.

This function is a shortcut to `masked_where`, with *condition* = (*x* > *value*).

**See also:**

[\*masked\\_where\*](#) Mask where a condition is met.

## Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
masked_array(data = [0 1 2 --],
             mask = [False False False  True],
             fill_value=999999)
```

`dask.array.ma.masked_greater_equal` (*x*, *value*, *copy=True*)

Mask an array where greater than or equal to a given value.

This function is a shortcut to `masked_where`, with *condition* = (*x* >= *value*).

**See also:**

[\*masked\\_where\*](#) Mask where a condition is met.

## Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data = [0 1 -- --],
             mask = [False False  True  True],
             fill_value=999999)
```

`dask.array.ma.masked_inside` (*x*, *v1*, *v2*, *copy=True*)

Mask an array inside a given interval.

Shortcut to `masked_where`, where *condition* is True for *x* inside the interval [*v1*,*v2*] ( $v1 \leq x \leq v2$ ). The boundaries *v1* and *v2* can be given in either order.

**See also:**

[`masked\_where`](#) Mask where a condition is met.

## Notes

The array *x* is prefilled with its filling value.

## Examples

```
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
             fill_value=1e+20)
```

The order of *v1* and *v2* doesn't matter.

```
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
             fill_value=1e+20)
```

`dask.array.ma.masked_invalid(a, copy=True)`

Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to `masked_where`, with *condition* = `~(np.isfinite(a))`. Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array\_like object.

**See also:**

[`masked\_where`](#) Mask where a condition is met.

## Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=np.float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0.,  1., NaN,  Inf,  4.])
>>> ma.masked_invalid(a)
masked_array(data = [0.0 1.0 -- -- 4.0],
             mask = [False False True True False],
             fill_value=1e+20)
```

`dask.array.ma.masked_less(x, value, copy=True)`

Mask an array where less than a given value.

This function is a shortcut to `masked_where`, with *condition* =  $(x < \text{value})$ .

**See also:**

[\*masked\\_where\*](#) Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less(a, 2)
masked_array(data = [-- -- 2 3],
             mask = [ True  True False False],
             fill_value=999999)
```

`dask.array.ma.masked_less_equal(x, value, copy=True)`

Mask an array where less than or equal to a given value.

This function is a shortcut to `masked_where`, with *condition* =  $(x \leq \text{value})$ .

**See also:**

[\*masked\\_where\*](#) Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less_equal(a, 2)
masked_array(data = [-- -- -- 3],
             mask = [ True  True  True False],
             fill_value=999999)
```

`dask.array.ma.masked_not_equal(x, value, copy=True)`

Mask an array where *not* equal to a given value.

This function is a shortcut to `masked_where`, with *condition* =  $(x \neq \text{value})$ .

**See also:**

[\*masked\\_where\*](#) Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
masked_array(data = [-- -- 2 --],
```

```
mask = [ True  True False  True],
fill_value=999999)
```

`dask.array.ma.masked_outside(x, v1, v2, copy=True)`

Mask an array outside a given interval.

Shortcut to `masked_where`, where *condition* is True for *x* outside the interval `[v1,v2]` ( $x < v1$  or  $x > v2$ ). The boundaries *v1* and *v2* can be given in either order.

**See also:**

[`masked\_where`](#) Mask where a condition is met.

## Notes

The array *x* is prefilled with its filling value.

## Examples

```
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
             mask = [ True  True False False  True  True],
             fill_value=1e+20)
```

The order of *v1* and *v2* doesn't matter.

```
>>> ma.masked_outside(x, 0.3, -0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
             mask = [ True  True False False  True  True],
             fill_value=1e+20)
```

`dask.array.ma.masked_values(x, value, rtol=1e-05, atol=1e-08, copy=True, shrink=True)`

Mask using floating point equality.

Return a MaskedArray, masked where the data in array *x* are approximately equal to *value*, i.e. where the following condition is True

$$(\text{abs}(x - \text{value}) \leq \text{atol} + \text{rtol} * \text{abs}(\text{value}))$$

The *fill\_value* is set to *value* and the mask is set to `nomask` if possible. For integers, consider using `masked_equal`.

**Parameters** *x*: array\_like

Array to mask.

**value**: float

Masking value.

**rtol**: float, optional

Tolerance parameter.

**atol**: float, optional

Tolerance parameter (1e-8).

**copy** : bool, optional

Whether to return a copy of *x*.

**shrink** : bool, optional

Whether to collapse a mask full of False to `nomask`.

**Returns result** : MaskedArray

The result of masking *x* where approximately equal to *value*.

**See also:**

[`masked\_where`](#) Mask where a condition is met.

[`masked\_equal`](#) Mask where equal to a given value (integers).

## Examples

```
>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data = [1.0 -- 2.0 -- 3.0],
             mask = [False True False True False],
             fill_value=1.1)
```

Note that *mask* is set to `nomask` if possible.

```
>>> ma.masked_values(x, 1.5)
masked_array(data = [ 1.  1.1  2.  1.1  3. ],
             mask = False,
             fill_value=1.5)
```

For integers, the fill value will be different in general to the result of `masked_equal`.

```
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
>>> ma.masked_values(x, 2)
masked_array(data = [0 1 -- 3 4],
             mask = [False False True False False],
             fill_value=2)
>>> ma.masked_equal(x, 2)
masked_array(data = [0 1 -- 3 4],
             mask = [False False True False False],
             fill_value=999999)
```

`dask.array.ma.masked_where` (*condition*, *a*, *copy=True*)

Mask an array where a condition is met.

Return *a* as an array masked where *condition* is True. Any masked values of *a* or *condition* are also masked in the output.

**Parameters condition** : array\_like

Masking condition. When *condition* tests floating point values for equality, consider using `masked_values` instead.

**a** : array\_like

Array to mask.

**copy** : bool

If True (default) make a copy of *a* in the result. If False modify *a* in place and return a view.

**Returns result** : MaskedArray

The result of masking *a* where *condition* is True.

**See also:**

[\*masked\\_values\*](#) Mask using floating point equality.

[\*masked\\_equal\*](#) Mask where equal to a given value.

[\*masked\\_not\\_equal\*](#) Mask where *not* equal to a given value.

[\*masked\\_less\\_equal\*](#) Mask where less than or equal to a given value.

[\*masked\\_greater\\_equal\*](#) Mask where greater than or equal to a given value.

[\*masked\\_less\*](#) Mask where less than a given value.

[\*masked\\_greater\*](#) Mask where greater than a given value.

[\*masked\\_inside\*](#) Mask inside a given interval.

[\*masked\\_outside\*](#) Mask outside a given interval.

[\*masked\\_invalid\*](#) Mask invalid values (NaNs or infs).

## Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data = [-- -- -- 3],
             mask = [ True  True  True False],
             fill_value=999999)
```

Mask array *b* conditional on *a*.

```
>>> b = ['a', 'b', 'c', 'd']
>>> ma.masked_where(a == 2, b)
masked_array(data = [a b -- d],
             mask = [False False  True False],
             fill_value=N/A)
```

Effect of the *copy* argument.

```
>>> c = ma.masked_where(a <= 2, a)
>>> c
masked_array(data = [-- -- -- 3],
             mask = [ True  True  True False],
             fill_value=999999)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
```

```
mask = [False True True False],
fill_value=999999)
>>> a
array([0, 1, 2, 3])
>>> c = ma.masked_where(a <= 2, a, copy=False)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
             mask = [False True True False],
             fill_value=999999)
>>> a
array([99, 1, 2, 3])
```

When *condition* or *a* contain masked values.

```
>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
>>> a
masked_array(data = [0 1 -- 3],
             mask = [False False True False],
             fill_value=999999)
>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)
>>> b
masked_array(data = [-- 1 2 3],
             mask = [ True False False False],
             fill_value=999999)
>>> ma.masked_where(a == 3, b)
masked_array(data = [-- 1 -- --],
             mask = [ True False True True],
             fill_value=999999)
```

`dask.array.ma.set_fill_value(a, fill_value)`

Set the filling value of *a*, if *a* is a masked array.

This function changes the fill value of the masked array *a* in place. If *a* is not a masked array, the function returns silently, without doing anything.

**Parameters** *a* : array\_like

Input array.

**fill\_value** : dtype

Filling value. A consistency test is performed to make sure the value is compatible with the dtype of *a*.

**Returns** None

Nothing returned by this function.

**See also:**

**maximum\_fill\_value** Return the default fill value for a dtype.

**MaskedArray.fill\_value** Return current fill value.

**MaskedArray.set\_fill\_value** Equivalent method.

## Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a = ma.masked_where(a < 3, a)
>>> a
masked_array(data = [-- -- -- 3 4],
             mask = [ True  True  True False False],
             fill_value=999999)
>>> ma.set_fill_value(a, -999)
>>> a
masked_array(data = [-- -- -- 3 4],
             mask = [ True  True  True False False],
             fill_value=-999)
```

Nothing happens if *a* is not a masked array.

```
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
>>> ma.set_fill_value(a, 100)
>>> a
[0, 1, 2, 3, 4]
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> ma.set_fill_value(a, 100)
>>> a
array([0, 1, 2, 3, 4])
```

`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’, or an array value. Such a value will fill the boundary with that value.

**The depth input 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.**

**Axes missing from this input will not be overlapped.**

## 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, asarray=True, fancy=True, getitem=None)`

Create dask array from something that looks like an array

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

**Parameters** `x` : array\_like

**chunks** : int, tuple

How to chunk the array. 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)).

-1 as a blocksize indicates the size of the corresponding dimension.

**name** : str, optional

The key name to use for the array. Defaults to a hash of `x`. Use `name=False` to generate a random name instead of hashing (fast)

**lock** : bool or Lock, optional

If `x` doesn't support concurrent reads then provide a lock here, or pass in `True` to have `dask.array` create one for you.

**asarray** : bool, optional

If `True` (default), then chunks will be converted to instances of `ndarray`. Set to `False` to pass passed chunks through unchanged.

**fancy** : bool, optional

If `x` doesn't support fancy indexing (e.g. indexing with lists or arrays) then set to False.  
Default is True.

## 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, 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 delayed
>>> value = delayed(np.ones)(5)
>>> array = from_delayed(value, (5,), float)
>>> array
dask.array<from-value, 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, regions=None, compute=True, return\_stored=False, \*\*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.

**regions: tuple of slices or iterable of tuple of slices**

Each `region` tuple in `regions` should be such that `target[region].shape = source.shape` for the corresponding source and target in `sources` and `targets`, respectively.

**compute: boolean, optional**

If true compute immediately, return `dask.delayed.Delayed` otherwise

**return\_stored: boolean, optional**

Optionally return the stored result (default `False`).

## 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 .npz files

This partitions the dask.array along one axis and stores each block along that axis as a single .npz 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)
```

```
$ tree data/ data/ |- 0.npz |- 1.npz |- 2.npz |- info
```

The .npz 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_wrap` (*fft\_func*, *kind=None*, *dtype=None*)

Wrap 1D complex FFT functions

Takes a function that behaves like `numpy.fft` functions and a specified kind to match it to that are named after the functions in the `numpy.fft` API.

Supported kinds include:

- `fft`
- `ifft`
- `rfft`
- `irfft`
- `hfft`
- `ihfft`

**Examples**

```
>>> parallel_fft = fft_wrap(np.fft.fft)
>>> parallel_ifft = fft_wrap(np.fft.ifft)
```

`dask.array.fft.fft` (*a*, *n=None*, *axis=None*)

Wrapping of `numpy.fft.fftpack.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.fftpack.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.

`norm` : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**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.fft2(a, s=None, axes=None)`

Wrapping of `numpy.fft.fftpack.fft2`

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.fftpack.fft2` docstring follows below:

Compute the 2-dimensional discrete Fourier Transform

This function computes the  $n$ -dimensional discrete Fourier Transform over any axes in an  $M$ -dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

**Parameters** **a** : array\_like

Input array, can be complex

**s** : sequence of ints, optional

Shape (length of each transformed axis) of the output ( $s[0]$  refers to axis 0,  $s[1]$  to axis 1, etc.). This corresponds to  $n$  for  $fft(x, n)$ . Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if  $s$  is not given, the shape of the input along the axes specified by *axes* is used.

**axes** : sequence of ints, optional

Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in *axes* means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

**norm** : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns** **out** : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by *axes*, or the last two axes if *axes* is not given.

#### Raises `ValueError`

If *s* and *axes* have different length, or *axes* not given and  $\text{len}(s) \neq 2$ .

#### `IndexError`

If an element of *axes* is larger than than the number of axes of *a*.

#### See also:

`numpy.fft` Overall view of discrete Fourier transforms, with definitions and conventions used.

`ifft2` The inverse two-dimensional FFT.

`fft` The one-dimensional FFT.

`fftn` The *n*-dimensional FFT.

`fftshift` Shifts zero-frequency terms to the center of the array. For two-dimensional input, swaps first and third quadrants, and second and fourth quadrants.

#### Notes

`fft2` is just `fftn` with a different default for *axes*.

The output, analogously to `fft`, contains the term for zero frequency in the low-order corner of the transformed axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of the axes, in order of decreasingly negative frequency.

See `fftn` for details and a plotting example, and `numpy.fft` for definitions and conventions used.

#### Examples

```
>>> a = np.mgrid[:5, :5][0]
>>> np.fft.fft2(a)
array([[ 50.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [ 0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [-12.5+17.20477401j,  0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [ 0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [-12.5 +4.0614962j,   0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [ 0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [-12.5 -4.0614962j,   0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [ 0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [-12.5-17.20477401j,  0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j],
       [ 0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j,          0.0 +0.j]])
```

```
dask.array.fft.fftn(a, s=None, axes=None)
```

Wrapping of `numpy.fft.fftpack.fftn`

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.fftpack.fftn` docstring follows below:

Compute the N-dimensional discrete Fourier Transform.

This function computes the  $N$ -dimensional discrete Fourier Transform over any number of axes in an  $M$ -dimensional array by means of the Fast Fourier Transform (FFT).

**Parameters** **a** : array\_like

Input array, can be complex.

**s** : sequence of ints, optional

Shape (length of each transformed axis) of the output ( $s[0]$  refers to axis 0,  $s[1]$  to axis 1, etc.). This corresponds to  $n$  for  $\text{fft}(x, n)$ . Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If  $s$  is not given, the shape of the input along the axes specified by *axes* is used.

**axes** : sequence of ints, optional

Axes over which to compute the FFT. If not given, the last  $\text{len}(s)$  axes are used, or all axes if  $s$  is also not specified. Repeated indices in *axes* means that the transform over that axis is performed multiple times.

**norm** : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns** **out** : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of  $s$  and  $a$ , as explained in the parameters section above.

**Raises** **ValueError**

If  $s$  and *axes* have different length.

**IndexError**

If an element of *axes* is larger than than the number of axes of  $a$ .

**See also:**

`numpy.fft` Overall view of discrete Fourier transforms, with definitions and conventions used.

`ifftn` The inverse of `fftn`, the inverse  $n$ -dimensional FFT.

`fft` The one-dimensional FFT, with definitions and conventions used.

`rfftn` The  $n$ -dimensional FFT of real input.

`fft2` The two-dimensional FFT.

`fftshift` Shifts zero-frequency terms to centre of array

## Notes

The output, analogously to `fft`, contains the term for zero frequency in the low-order corner of all axes, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

See `numpy.fft` for details, definitions and conventions used.

## Examples

```

>>> a = np.mgrid[:3, :3, :3][0]
>>> np.fft.fftn(a, axes=(1, 2))
array([[[ 0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [[ 9.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [[ 18.+0.j, 0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]])
>>> np.fft.fftn(a, (2, 2), axes=(0, 1))
array([[[ 2.+0.j,  2.+0.j,  2.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [[-2.+0.j, -2.+0.j, -2.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]])

```

```

>>> import matplotlib.pyplot as plt
>>> [X, Y] = np.meshgrid(2 * np.pi * np.arange(200) / 12,
...                     2 * np.pi * np.arange(200) / 34)
>>> S = np.sin(X) + np.cos(Y) + np.random.uniform(0, 1, X.shape)
>>> FS = np.fft.fftn(S)
>>> plt.imshow(np.log(np.abs(np.fft.fftshift(FS)**2)))
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()

```

`dask.array.fft.iff` (*a*, *n=None*, *axis=None*)

Wrapping of `numpy.fft.fftpack.iff`

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.fftpack.iff` 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, `iff(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]` should contain the positive-frequency terms,
- `a[n//2 + 1:]` should contain the negative-frequency terms, in increasing order starting from the most negative frequency.

For an even number of input points, `A[n//2]` represents the sum of the values at the positive and negative Nyquist frequencies, as the two are aliased together. 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.

**norm** : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**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--')
...
>>> plt.legend(('real', 'imaginary'))
...
>>> plt.show()
```

dask.array.fft.**ifft2** (*a*, *s=None*, *axes=None*)  
Wrapping of numpy.fft.fftpack.ifft2

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.fftpack.ifft2` docstring follows below:

Compute the 2-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the 2-dimensional discrete Fourier Transform over any number of axes in an  $M$ -dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifft2(fft2(a)) == a` to within numerical accuracy. By default, the inverse transform is computed over the last two axes of the input array.

The input, analogously to `ifft`, should be ordered in the same way as is returned by `fft2`, i.e. it should have the term for zero frequency in the low-order corner of the two axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of both axes, in order of decreasingly negative frequency.

**Parameters** `a` : array\_like

Input array, can be complex.

`s` : sequence of ints, optional

Shape (length of each axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to  $n$  for `ifft(x, n)`. Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If `s` is not given, the shape of the input along the axes specified by `axes` is used. See notes for issue on `ifft` zero padding.

`axes` : sequence of ints, optional

Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in `axes` means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

`norm` : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns** `out` : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by `axes`, or the last two axes if `axes` is not given.

**Raises** `ValueError`

If `s` and `axes` have different length, or `axes` not given and `len(s) != 2`.

**IndexError**

If an element of `axes` is larger than than the number of axes of `a`.

**See also:**

`numpy.fft` Overall view of discrete Fourier transforms, with definitions and conventions used.

`fft2` The forward 2-dimensional FFT, of which `ifft2` is the inverse.

`ifftn` The inverse of the  $n$ -dimensional FFT.

`fft` The one-dimensional FFT.

`ifft` The one-dimensional inverse FFT.

## Notes

`ifft2` is just `ifftn` with a different default for `axes`.

See `ifftn` for details and a plotting example, and `numpy.fft` for definition and conventions used.

Zero-padding, analogously with `ifft`, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before `ifft2` is called.

## Examples

```
>>> a = 4 * np.eye(4)
>>> np.fft.ifft2(a)
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j]])
```

`dask.array.fft.ifftn(a, s=None, axes=None)`

Wrapping of `numpy.fft.fftpack.ifftn`

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.fftpack.ifftn` docstring follows below:

Compute the N-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifftn(fftn(a)) == a` to within numerical accuracy. For a description of the definitions and conventions used, see `numpy.fft`.

The input, analogously to `ifft`, should be ordered in the same way as is returned by `fftn`, i.e. it should have the term for zero frequency in all axes in the low-order corner, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

**Parameters** `a` : array\_like

Input array, can be complex.

`s` : sequence of ints, optional

Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to `n` for `ifft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if `s` is not given, the shape of the input along the axes specified by `axes` is used. See notes for issue on `ifft` zero padding.

`axes` : sequence of ints, optional

Axes over which to compute the IFFT. If not given, the last `len(s)` axes are used, or all axes if `s` is also not specified. Repeated indices in `axes` means that the inverse transform over that axis is performed multiple times.

`norm` : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns out** : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* or *a*, as explained in the parameters section above.

**Raises ValueError**

If *s* and *axes* have different length.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

**numpy.fft** Overall view of discrete Fourier transforms, with definitions and conventions used.

**fftn** The forward *n*-dimensional FFT, of which *ifftn* is the inverse.

**ifft** The one-dimensional inverse FFT.

**ifft2** The two-dimensional inverse FFT.

**ifftshift** Undoes *fftshift*, shifts zero-frequency terms to beginning of array.

## Notes

See *numpy.fft* for definitions and conventions used.

Zero-padding, analogously with *ifft*, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before *ifftn* is called.

## Examples

```
>>> a = np.eye(4)
>>> np.fft.ifftn(np.fft.fftn(a, axes=(0,)), axes=(1,))
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]])
```

Create and plot an image with band-limited frequency content:

```
>>> import matplotlib.pyplot as plt
>>> n = np.zeros((200,200), dtype=complex)
>>> n[60:80, 20:40] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20, 20)))
>>> im = np.fft.ifftn(n).real
>>> plt.imshow(im)
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()
```

`dask.array.fft.rfft(a, n=None, axis=None)`

Wrapping of `numpy.fft.fftpack.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.fftpack.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.

**norm** : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**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 Hermitian-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  $A = \text{rfft}(a)$  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.**rfft2** (*a*, *s=None*, *axes=None*)

Wrapping of numpy.fft.fftpack.rfft2

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.fftpack.rfft2` docstring follows below:

Compute the 2-dimensional FFT of a real array.

**Parameters** *a* : array

Input array, taken to be real.

*s* : sequence of ints, optional

Shape of the FFT.

*axes* : sequence of ints, optional

Axes over which to compute the FFT.

*norm* : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns** *out* : ndarray

The result of the real 2-D FFT.

**See also:**

[\*rfftn\*](#) Compute the N-dimensional discrete Fourier Transform for real input.

## Notes

This is really just *rfftn* with different default behavior. For more details see *rfftn*.

dask.array.fft.**rfftn** (*a*, *s=None*, *axes=None*)

Wrapping of numpy.fft.fftpack.rfftn

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.fftpack.rfftn` docstring follows below:

Compute the N-dimensional discrete Fourier Transform for real input.

This function computes the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional real array by means of the Fast Fourier Transform (FFT). By default, all axes are transformed, with the real transform performed over the last axis, while the remaining transforms are complex.

**Parameters** *a* : array\_like

Input array, taken to be real.

*s* : sequence of ints, optional

Shape (length along each transformed axis) to use from the input. (*s* [0] refers to axis 0, *s* [1] to axis 1, etc.). The final element of *s* corresponds to *n* for `rffft(x, n)`, while for the remaining axes, it corresponds to *n* for `fft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used.

*axes* : sequence of ints, optional

Axes over which to compute the FFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified.

*norm* : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns** *out* : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* and *a*, as explained in the parameters section above. The length of the last axis transformed will be `s[-1] // 2 + 1`, while the remaining transformed axes will have lengths according to *s*, or unchanged from the input.

**Raises** **ValueError**

If *s* and *axes* have different length.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

`irfftn` The inverse of `rfftn`, i.e. the inverse of the n-dimensional FFT of real input.

`fft` The one-dimensional FFT, with definitions and conventions used.

`rfft` The one-dimensional FFT of real input.

`fftn` The n-dimensional FFT.

`rfft2` The two-dimensional FFT of real input.

**Notes**

The transform for real input is performed over the last transformation axis, as by `rfft`, then the transform over the remaining axes is performed as by `fftn`. The order of the output is as for `rfft` for the final transformation axis, and as for `fftn` for the remaining transformation axes.

See `fft` for details, definitions and conventions used.

## Examples

```
>>> a = np.ones((2, 2, 2))
>>> np.fft.rfftn(a)
array([[[ 8.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j]],
       [[ 0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j]])
```

```
>>> np.fft.rfftn(a, axes=(2, 0))
array([[[ 4.+0.j,  0.+0.j],
        [ 4.+0.j,  0.+0.j]],
       [[ 0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j]])
```

`dask.array.fft.irfft(a, n=None, axis=None)`

Wrapping of `numpy.fft.fftpack.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.fftpack.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 Hermitian-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. If not given, the last axis is used.

`norm` : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**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 Hermitian-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.fft.irfft2(a, s=None, axes=None)`

Wrapping of `numpy.fft.fftpack.irfft2`

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.fftpack.irfft2` docstring follows below:

Compute the 2-dimensional inverse FFT of a real array.

**Parameters** *a* : array\_like

The input array

*s* : sequence of ints, optional

Shape of the inverse FFT.

*axes* : sequence of ints, optional

The axes over which to compute the inverse fft. Default is the last two axes.

*norm* : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**Returns out** : ndarray

The result of the inverse real 2-D FFT.

**See also:**

*irfftn* Compute the inverse of the N-dimensional FFT of real input.

## Notes

This is really *irfftn* with different defaults. For more details see *irfftn*.

`dask.array.fft.irfftn(a, s=None, axes=None)`

Wrapping of `numpy.fft.fftpack.irfftn`

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.fftpack.irfftn` docstring follows below:

Compute the inverse of the N-dimensional FFT of real input.

This function computes the inverse of the N-dimensional discrete Fourier Transform for real input over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `irfftn(rfftn(a), a.shape) == a` to within numerical accuracy. (The `a.shape` is necessary like `len(a)` is for *irfft*, and for the same reason.)

The input should be ordered in the same way as is returned by *rfftn*, i.e. as for *irfft* for the final transformation axis, and as for *ifftn* along all the other axes.

**Parameters a** : array\_like

Input array.

**s** : sequence of ints, optional

Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). *s* is also the number of input points used along this axis, except for the last axis, where `s[-1] // 2 + 1` points of the input are used. Along any axis, if the shape indicated by *s* is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If *s* is not given, the shape of the input along the axes specified by *axes* is used.

**axes** : sequence of ints, optional

Axes over which to compute the inverse FFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified. Repeated indices in *axes* means that the inverse transform over that axis is performed multiple times.

**norm** : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**Returns out** : ndarray

The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* or *a*, as explained in the parameters section above. The length of each transformed axis is as given by the corresponding element of *s*, or the length of the input in every axis except for the last one if *s* is not given. In the final transformed axis the length of the output when *s* is not given is  $2 * (m-1)$  where *m* is the length of the final transformed axis of the input. To get an odd number of output points in the final axis, *s* must be specified.

#### Raises `ValueError`

If *s* and *axes* have different length.

#### `IndexError`

If an element of *axes* is larger than than the number of axes of *a*.

#### See also:

`rfftn` The forward n-dimensional FFT of real input, of which `ifftn` is the inverse.

`fft` The one-dimensional FFT, with definitions and conventions used.

`irfft` The inverse of the one-dimensional FFT of real input.

`irfft2` The inverse of the two-dimensional FFT of real input.

#### Notes

See `fft` for definitions and conventions used.

See `rfft` for definitions and conventions used for real input.

#### Examples

```
>>> a = np.zeros((3, 2, 2))
>>> a[0, 0, 0] = 3 * 2 * 2
>>> np.fft.irfftn(a)
array([[[ 1.,  1.],
        [ 1.,  1.]],
       [[ 1.,  1.],
        [ 1.,  1.]],
       [[ 1.,  1.],
        [ 1.,  1.]])
```

`dask.array.fft.hfft` (*a*, *n=None*, *axis=None*)

Wrapping of `numpy.fft.fftpack.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.fftpack.hfft` docstring follows below:

Compute the FFT of a signal which has Hermitian symmetry (real spectrum).

**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 FFT. If not given, the last axis is used.

**norm** : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**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:**

*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 has Hermitian symmetry in the time domain and is real in the frequency 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, 2, 3, 4, 3, 2])
>>> np.fft.fft(signal)
array([ 15.+0.j, -4.+0.j,  0.+0.j, -1.-0.j,  0.+0.j, -4.+0.j])
>>> np.fft.hfft(signal[:4]) # Input first half of signal
array([ 15., -4.,  0., -1.,  0., -4.])
>>> np.fft.hfft(signal, 6) # Input entire signal and truncate
array([ 15., -4.,  0., -1.,  0., -4.])
```

```
>>> 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=None)`

Wrapping of `numpy.fft.fftpack.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.fftpack.ihfft` docstring follows below:

Compute the inverse FFT of a signal which has Hermitian symmetry.

**Parameters** **a** : array\_like

Input array.

**n** : int, optional

Length of the inverse FFT. 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 inverse FFT. If not given, the last axis is used.

**norm** : {None, "ortho"}, optional

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**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  $(n+1) / 2$ .

**See also:**

`hfft`, `irfft`

## Notes

`hfft/ihfft` are a pair analogous to `rfft/irfft`, but for the opposite case: here the signal has Hermitian symmetry in the time domain and is real in the frequency 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

```
>>> spectrum = np.array([ 15, -4, 0, -1, 0, -4])
>>> np.fft.ifft(spectrum)
array([ 1.+0.j,  2.-0.j,  3.+0.j,  4.+0.j,  3.+0.j,  2.-0.j])
>>> np.fft.ihfft(spectrum)
array([ 1.-0.j,  2.-0.j,  3.-0.j,  4.-0.j])
```

`dask.array.fft.fftfreq(n, d=1.0)`

Return the Discrete Fourier Transform sample frequencies.

The returned float array *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length  $n$  and a sample spacing  $d$ :

```
f = [0, 1, ..., n/2-1, -n/2, ..., -1] / (d*n)  if n is even
f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)  if n is odd
```

**Parameters**  $n$  : int

Window length.

$d$  : scalar, optional

Sample spacing (inverse of the sampling rate). Defaults to 1.

**Returns**  $f$  : ndarray

Array of length  $n$  containing the sample frequencies.

## Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0. ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

`dask.array.fft.rfftfreq`( $n, d=1.0$ )

Return the Discrete Fourier Transform sample frequencies (for usage with `rfft`, `irfft`).

The returned float array  $f$  contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length  $n$  and a sample spacing  $d$ :

```
f = [0, 1, ..., n/2-1, n/2] / (d*n)  if n is even
f = [0, 1, ..., (n-1)/2-1, (n-1)/2] / (d*n)  if n is odd
```

Unlike `fftfreq` (but like `scipy.fftpack.rfftfreq`) the Nyquist frequency component is considered to be positive.

**Parameters**  $n$  : int

Window length.

$d$  : scalar, optional

Sample spacing (inverse of the sampling rate). Defaults to 1.

**Returns**  $f$  : ndarray

Array of length  $n//2 + 1$  containing the sample frequencies.

## Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5, -3, 4], dtype=float)
>>> fourier = np.fft.rfft(signal)
>>> n = signal.size
>>> sample_rate = 100
>>> freq = np.fft.rfftfreq(n, d=1./sample_rate)
```

```

>>> freq
array([ 0., 10., 20., 30., 40., -50., -40., -30., -20., -10.])
>>> freq = np.fft.rfftfreq(n, d=1./sample_rate)
>>> freq
array([ 0., 10., 20., 30., 40., 50.])

```

`dask.array.fft.fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

**Parameters** `x` : array\_like

Input array.

**axes** : int or shape tuple, optional

Axes over which to shift. Default is `None`, which shifts all axes.

**Returns** `y` : ndarray

The shifted array.

**See also:**

[`ifftshift`](#) The inverse of `fftshift`.

## Examples

```

>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0., 1., 2., 3., 4., -5., -4., -3., -2., -1.])
>>> np.fft.fftshift(freqs)
array([-5., -4., -3., -2., -1., 0., 1., 2., 3., 4.])

```

Shift the zero-frequency component only along the second axis:

```

>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0., 1., 2.],
       [ 3., 4., -4.],
       [-3., -2., -1.]])
>>> np.fft.fftshift(freqs, axes=(1,))
array([[ 2., 0., 1.],
       [-4., 3., 4.],
       [-1., -3., -2.]])

```

`dask.array.fft.ifftshift(x, axes=None)`

The inverse of `fftshift`. Although identical for even-length `x`, the functions differ by one sample for odd-length `x`.

**Parameters** `x` : array\_like

Input array.

**axes** : int or shape tuple, optional

Axes over which to calculate. Defaults to `None`, which shifts all axes.

**Returns** `y` : ndarray

The shifted array.

**See also:**

`fftshift` Shift zero-frequency component to the center of the spectrum.

### Examples

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.ifftshift(np.fft.fftshift(freqs))
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
```

`dask.array.random.beta` (*a*, *b*, *size=None*)

Draw samples from a Beta distribution.

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** : 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. Default is None, in which case a single value is returned.

**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  $\geq 0$  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,  $\geq 0$ .

**p** : float

parameter,  $\geq 0$  and  $\leq 1$ .

**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. Default is None, in which case a single value is returned.

**Returns samples** : ndarray or 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  $p*n \leq 5$ , 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

[R123], [R124], [R125], [R126], [R127]

## 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** : 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. Default is None, in which case a single value 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:

$$Q = \sum_{i=0}^{df} X_i^2$$

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  $\Gamma$  is the gamma function,

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt.$$

## References

[R128]

## Examples

```
>> np.random.chisquare(2,4) array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

```
dask.array.random.exponential(scale=1.0, size=None)
```

Draw samples from an 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.  $\beta$  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 [R131].

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 [R129], or the time between page requests to Wikipedia [R130].

**Parameters** `scale` : float

The scale parameter,  $\beta = 1/\lambda$ .

**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. Default is None, in which case a single value is returned.

## References

[R129], [R130], [R131]

`dask.array.random.f` (*dfnum*, *dfden*, *size=None*)

Draw samples from an 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** : 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. Default is None, in which case a single value is returned.

**Returns** `samples` : ndarray or 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

[R132], [R133]

## 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** : 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. Default is None, in which case a single value is returned.

**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  $\Gamma$  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

[R134], [R135]

## 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,  $k = 1, 2, \dots$

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** : 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. Default is None, in which case a single value is returned.

**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)
```

Draw samples from a 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** : 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. Default is None, in which case a single value is returned.

**Returns samples** : ndarray or scalar

**See also:**

`scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`, `scipy.stats.genextreme`, `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  $\mu$  is the mode, a location parameter, and  $\beta$  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 Fréchet.

The function has a mean of  $\mu + 0.57721\beta$  and a variance of  $\frac{\pi^2}{6}\beta^2$ .

## References

[R136], [R137]

## 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.sqrt(6) / np.pi >> 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 ints, optional

Output shape. If the given shape is, e.g., (*m*, *n*, *k*), then *m* \* *n* \* *k* samples are drawn. Default is None, in which case a single value is returned.

**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,  $n = \text{ngood}$ ,  $m = \text{nbad}$ , and  $N = \text{number of samples}$ .

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

[R138], [R139], [R140]

## 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, optional

The position,  $\mu$ , of the distribution peak.

**scale** : float, optional

$\lambda$ , the exponential decay.

**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. Default is None, in which case a single value is returned.

**Returns** **samples** : ndarray or float

## 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

[R141], [R142], [R143], [R144]

## 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` : 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. Default is `None`, in which case a single value is returned.

**Returns** `samples` : ndarray or 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  $\mu$  = 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

[R145], [R146], [R147]

## 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, log-
gist(bins, loc, scale)*count.max()/.. logist(bins, loc, scale).max()) >> plt.show()
```

```
dask.array.random.lognormal (mean=0.0, sigma=1.0, size=None)
```

Draw samples 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** : 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. Default is None, in which case a single value is returned.

**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  $\mu$  is the mean and  $\sigma$  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

[R148], [R149]

## 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='mid') >> 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 shape parameter,  $0 < p < 1$ .

**Parameters** `loc` : float

**scale** : float > 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. Default is None, in which case a single value is returned.

**Returns** `samples` : ndarray or 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

[R150], [R151], [R152], [R153]

## 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,  $\geq 0$  and  $\leq 1$ .

**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. Default is None, in which case a single value is returned.

**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

[R154], [R155]

## 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  $\chi^2$  distribution is a generalisation of the  $\chi^2$  distribution.

**Parameters** *df* : int

Degrees of freedom, should be > 0 as of Numpy 1.10, should be > 1 for earlier versions.

**nonc** : float

Non-centrality, should be non-negative.

**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. Default is None, in which case a single value is returned.

## Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2} (nonc/2)^i}{i!} \mathbb{1}_{Y_{df+2i}}(x),$$

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

[R156], [R157]

## 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, optional

Output shape. If the given shape is, e.g., (*m*, *n*, *k*), then *m* \* *n* \* *k* samples are drawn. Default is None, in which case a single value is returned.

**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

[R158], [R159]

## 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 [R161], 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 [R161].

**Parameters** `loc` : float

Mean (“centre”) of the distribution.

**scale** : float

Standard deviation (spread or “width”) of the distribution.

**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. Default is None, in which case a single value is returned.

**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  $\mu$  is the mean and  $\sigma$  the standard deviation. The square of the standard deviation,  $\sigma^2$ , is called the variance.

The function has its peak at the mean, and its “spread” increases with the standard deviation (the function reaches 0.607 times its maximum at  $x + \sigma$  and  $x - \sigma$  [R161]). This implies that `numpy.random.normal` is more likely to return samples lying close to the mean, rather than those far away.

## References

[R160], [R161]

## 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 1 and multiplying by the scale parameter *m* (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is *mu*, where the standard Pareto distribution has location *mu* = 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* : 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. Default is None, in which case a single value is returned.

**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 scale.

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 [R162]. It is one of the so-called “fat-tailed” distributions.

## References

[R162], [R163], [R164], [R165]

## Examples

Draw samples from the distribution:

```
>> a, m = 3., 2. # shape and mode >> s = (np.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>> import matplotlib.pyplot as plt >> count, bins, _ = plt.hist(s, 100, normed=True) >> 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 or sequence of float

Expectation of interval, should be  $\geq 0$ . A sequence of expectation intervals must be broadcastable over the requested size.

**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. Default is None, in which case a single value is returned.

**Returns** **samples** : ndarray or scalar

The drawn samples, of shape *size*, if it was provided.

## Notes

The Poisson distribution

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation  $\lambda$  the Poisson distribution  $f(k; \lambda)$  describes the probability of *k* events occurring within the observed interval  $\lambda$ .

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

[R166], [R167]

## 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()
```

Draw each 100 values for lambda 100 and 500:

```
>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))
```

`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, &gt; 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. Default is None, in which case a single value is returned.

**Returns** `samples` : ndarray or 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**[\[R168\]](#), [\[R169\]](#)**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.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

Output shape. If the given shape is, e.g.,  $(m, n, k)$ , then  $m * n * k$  samples are drawn. Default is `None`, in which case a single value is returned.

**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

Output shape. If the given shape is, e.g.,  $(m, n, k)$ , then  $m * n * k$  samples are drawn. Default is `None`, in which case a single value is returned.

**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  $\chi$  and Weibull distributions are generalizations of the Rayleigh.

**Parameters** `scale` : scalar

Scale, also equals the mode. Should be  $\geq 0$ .

`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. Default is None, in which case a single value is returned.

## Notes

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{-\frac{x^2}{scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

## References

[R170], [R171]

## 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*)

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

**Parameters** `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. Default is None, in which case a single value is returned.

**Returns** `samples` : ndarray or scalar

The drawn samples.

## Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + \left(\frac{x-x_0}{\gamma}\right)^2\right]}$$

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

[R172], [R173], [R174]

## 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, optional

Output shape. If the given shape is, e.g., (m, n, k), then m \* n \* k samples are drawn. Default is None, in which case a single value is returned.

**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, optional

Output shape. If the given shape is, e.g.,  $(m, n, k)$ , then  $m * n * k$  samples are drawn. Default is `None`, in which case a single value is returned.

**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  $\theta$  the scale, and  $\Gamma$  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

[R175], [R176]

## 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*)

Draw samples from a standard Normal distribution (mean=0, stdev=1).

**Parameters** `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. 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)
```

Draw samples from a 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. If the given shape is, e.g., (*m*, *n*, *k*), then *m* \* *n* \* *k* samples are drawn. 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 first 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

[R177], [R178]

## Examples

From Dalgaard page 83 [R177], 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)) 0.009069999999999999 #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. If the given shape is, e.g., (*m, n, k*), then *m \* n \* k* samples are drawn. 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

$$P(x; l, m, r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

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

[R179]

## 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=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

**Parameters** **low** : float, 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

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value 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  $\geq 0$ .

**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. Default is None, in which case a single value is returned.

**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  $\mu$  is the mode and  $\kappa$  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

[R180], [R181]

## 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 >> from scipy.special import i0 >> plt.hist(s, 50, normed=True) >> x = np.linspace(-np.pi, np.pi, num=51) >> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa)) >> plt.plot(x, 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 equal to 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. If the given shape is, e.g., (m, n, k), then m \* n \* k samples are drawn. 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

$$P(x; \text{mean}, \text{scale}) = \sqrt{\frac{\text{scale}}{2\pi x^3}} e^{-\frac{\text{scale}(x - \text{mean})^2}{2 \cdot \text{mean}^2 x}}$$

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

[R182], [R183], [R184]

## 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*)

Draw samples from a 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  $\lambda$  is just  $X = \lambda(-\ln(U))^{1/a}$ .

**Parameters** *a* : float

Shape of the distribution.

**size** : int or tuple of ints, optionalOutput shape. If the given shape is, e.g., (*m*, *n*, *k*), then *m* \* *n* \* *k* samples are drawn. Default is None, in which case a single value is returned.**Returns** **samples** : ndarray**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

$$p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a},$$

where *a* is the shape and  $\lambda$  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**

[R185], [R186], [R187]

**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.stats.ttest_ind(a, b, axis=0, equal_var=True)
```

Calculates the T-test for the means of TWO INDEPENDENT samples of scores.

This is a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values. This test assumes that the populations have identical variances by default.

**Parameters** *a*, *b* : array\_like

The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

**axis** : int or None, optional

Axis along which to compute test. If None, compute over the whole arrays, *a*, and *b*.

**equal\_var** : bool, optional

If True (default), perform a standard independent 2 sample test that assumes equal population variances [R188]. If False, perform Welch's t-test, which does not assume equal population variance [R189]. .. versionadded:: 0.11.0

**nan\_policy** : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns** **statistic** : float or array

The calculated t-statistic.

**pvalue** : float or array

The two-tailed p-value.

## Notes

We can use this test, if we observe two independent samples from the same or different population, e.g. exam scores of boys and girls or of two ethnic groups. The test measures whether the average (expected) value differs significantly across samples. If we observe a large p-value, for example larger than 0.05 or 0.1, then we cannot reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages.

## References

[R188], [R189]

## Examples

```
>> from scipy import stats >> np.random.seed(12345678)
```

Test with sample with identical means:

```
>> rvs1 = stats.norm.rvs(loc=5, scale=10, size=500) >> rvs2 = stats.norm.rvs(loc=5, scale=10, size=500)
>> stats.ttest_ind(rvs1, rvs2) (0.26833823296239279, 0.78849443369564776) >> stats.ttest_ind(rvs1, rvs2,
equal_var = False) (0.26833823296239279, 0.78849452749500748)
```

*ttest\_ind* underestimates p for unequal variances:

```
>> rvs3 = stats.norm.rvs(loc=5, scale=20, size=500) >> stats.ttest_ind(rvs1, rvs3) (-0.46580283298287162,
0.64145827413436174) >> stats.ttest_ind(rvs1, rvs3, equal_var = False) (-0.46580283298287162,
0.64149646246569292)
```

When  $n1 \neq n2$ , the equal variance t-statistic is no longer equal to the unequal variance t-statistic:

```
>> rvs4 = stats.norm.rvs(loc=5, scale=20, size=100) >> stats.ttest_ind(rvs1, rvs4) (-0.99882539442782481,
0.3182832709103896) >> stats.ttest_ind(rvs1, rvs4, equal_var = False) (-0.69712570584654099,
0.48716927725402048)
```

T-test with different means, variance, and n:

```
>> rvs5 = stats.norm.rvs(loc=8, scale=20, size=100) >> stats.ttest_ind(rvs1, rvs5) (-1.4679669854490653,
0.14263895620529152) >> stats.ttest_ind(rvs1, rvs5, equal_var = False) (-0.94365973617132992,
0.34744170334794122)
```

```
dask.array.stats.ttest_1samp(a, popmean, axis=0, nan_policy='propagate')
```

Calculates the T-test for the mean of ONE group of scores.

This is a two-sided test for the null hypothesis that the expected value (mean) of a sample of independent observations *a* is equal to the given population mean, *popmean*.

**Parameters** *a* : array\_like

sample observation

**popmean** : float or array\_like

expected value in null hypothesis, if array\_like than it must have the same shape as *a* excluding the axis dimension

**axis** : int or None, optional

Axis along which to compute test. If None, compute over the whole array *a*.

**nan\_policy** : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns** *statistic* : float or array

t-statistic

**pvalue** : float or array

two-tailed p-value

## Examples

```
>> from scipy import stats
```

```
>> np.random.seed(7654567) # fix seed to get the same result >> rvs = stats.norm.rvs(loc=5, scale=10,
size=(50,2))
```

Test if mean of random sample is equal to true mean, and different mean. We reject the null hypothesis in the second case and don't reject it in the first case.

```
>> stats.ttest_1samp(rvs,5.0) (array([-0.68014479, -0.04323899]), array([ 0.49961383, 0.96568674])) >>
stats.ttest_1samp(rvs,0.0) (array([ 2.77025808, 4.11038784]), array([ 0.00789095, 0.00014999]))
```

Examples using axis and non-scalar dimension for population mean.

```
>> stats.ttest_1samp(rvs,[5.0,0.0]) (array([-0.68014479, 4.11038784]), array([ 4.99613833e-01, 1.49986458e-
04])) >> stats.ttest_1samp(rvs.T,[5.0,0.0],axis=1) (array([-0.68014479, 4.11038784]), array([ 4.99613833e-01,
1.49986458e-04])) >> stats.ttest_1samp(rvs,[[5.0],[0.0]]) (array([[[-0.68014479, -0.04323899],
[ 2.77025808, 4.11038784]]], array([[ 4.99613833e-01, 9.65686743e-01], [ 7.89094663e-03,
1.49986458e-04]]))
```

```
dask.array.stats.ttest_rel(a, b, axis=0, nan_policy='propagate')
```

Calculates the T-test on TWO RELATED samples of scores, a and b.

This is a two-sided test for the null hypothesis that 2 related or repeated samples have identical average (expected) values.

**Parameters** **a, b** : array\_like

The arrays must have the same shape.

**axis** : int or None, optional

Axis along which to compute test. If None, compute over the whole arrays, *a*, and *b*.

**nan\_policy** : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns** **statistic** : float or array

t-statistic

**pvalue** : float or array

two-tailed p-value

## Notes

Examples for the use are scores of the same set of student in different exams, or repeated sampling from the same units. The test measures whether the average score differs significantly across samples (e.g. exams). If we observe a large p-value, for example greater than 0.05 or 0.1 then we cannot reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages. Small p-values are associated with large t-statistics.

## References

[http://en.wikipedia.org/wiki/T-test#Dependent\\_t-test](http://en.wikipedia.org/wiki/T-test#Dependent_t-test)

## Examples

```
>> from scipy import stats >> np.random.seed(12345678) # fix random seed to get same numbers
>> rvs1 = stats.norm.rvs(loc=5,scale=10,size=500) >> rvs2 = (stats.norm.rvs(loc=5,scale=10,size=500)
+ .. stats.norm.rvs(scale=0.2,size=500)) >> stats.ttest_rel(rvs1,rvs2) (0.24101764965300962,
0.80964043445811562) >> rvs3 = (stats.norm.rvs(loc=8,scale=10,size=500) + ..
stats.norm.rvs(scale=0.2,size=500)) >> stats.ttest_rel(rvs1,rvs3) (-3.9995108708727933,
7.3082402191726459e-005)
```

```
dask.array.stats.chisquare (f_obs, f_exp=None, ddof=0, axis=0)
```

Calculates a one-way chi square test.

The chi square test tests the null hypothesis that the categorical data has the given frequencies.

**Parameters** **f\_obs** : array\_like

Observed frequencies in each category.

**f\_exp** : array\_like, optional

Expected frequencies in each category. By default the categories are assumed to be equally likely.

**ddof** : int, optional

“Delta degrees of freedom”: adjustment to the degrees of freedom for the p-value. The p-value is computed using a chi-squared distribution with  $k - 1 - \text{ddof}$  degrees of freedom, where  $k$  is the number of observed frequencies. The default value of *ddof* is 0.

**axis** : int or None, optional

The axis of the broadcast result of *f\_obs* and *f\_exp* along which to apply the test. If axis is None, all values in *f\_obs* are treated as a single data set. Default is 0.

**Returns** **chisq** : float or ndarray

The chi-squared test statistic. The value is a float if *axis* is None or *f\_obs* and *f\_exp* are 1-D.

**p** : float or ndarray

The p-value of the test. The value is a float if *ddof* and the return value *chisq* are scalars.

**See also:**

[\*power\\_divergence\*](#), `mstats.chisquare`

## Notes

This test is invalid when the observed or expected frequencies in each category are too small. A typical rule is that all of the observed and expected frequencies should be at least 5.

The default degrees of freedom,  $k-1$ , are for the case when no parameters of the distribution are estimated. If  $p$  parameters are estimated by efficient maximum likelihood then the correct degrees of freedom are  $k-1-p$ . If the parameters are estimated in a different way, then the dof can be between  $k-1-p$  and  $k-1$ . However, it is also possible that the asymptotic distribution is not a chisquare, in which case this test is not appropriate.

## References

[R190], [R191]

## Examples

When just *f\_obs* is given, it is assumed that the expected frequencies are uniform and given by the mean of the observed frequencies.

```
>> from scipy.stats import chisquare >> chisquare([16, 18, 16, 14, 12, 12]) (2.0, 0.84914503608460956)
```

With *f\_exp* the expected frequencies can be given.

```
>> chisquare([16, 18, 16, 14, 12, 12], f_exp=[16, 16, 16, 16, 16, 8]) (3.5, 0.62338762774958223)
```

When *f\_obs* is 2-D, by default the test is applied to each column.

```
>> obs = np.array([[16, 18, 16, 14, 12, 12], [32, 24, 16, 28, 20, 24]]).T >> obs.shape (6, 2) >> chisquare(obs) (array([ 2. , 6.66666667]), array([ 0.84914504, 0.24663415]))
```

By setting *axis=None*, the test is applied to all data in the array, which is equivalent to applying the test to the flattened array.

```
>> chisquare(obs, axis=None) (23.31034482758621, 0.015975692534127565) >> chisquare(obs.ravel())
(23.31034482758621, 0.015975692534127565)
```

*ddof* is the change to make to the default degrees of freedom.

```
>> chisquare([16, 18, 16, 14, 12, 12], ddof=1) (2.0, 0.73575888234288467)
```

The calculation of the p-values is done by broadcasting the chi-squared statistic with *ddof*.

```
>> chisquare([16, 18, 16, 14, 12, 12], ddof=[0,1,2]) (2.0, array([ 0.84914504, 0.73575888, 0.5724067 ]))
```

*f\_obs* and *f\_exp* are also broadcast. In the following, *f\_obs* has shape (6,) and *f\_exp* has shape (2, 6), so the result of broadcasting *f\_obs* and *f\_exp* has shape (2, 6). To compute the desired chi-squared statistics, we use *axis=1*:

```
>> chisquare([16, 18, 16, 14, 12, 12], .. f_exp=[[16, 16, 16, 16, 16, 8], [8, 20, 20, 16, 12, 12]], .. axis=1) (array([
3.5 , 9.25]), array([ 0.62338763, 0.09949846]))
```

`dask.array.stats.power_divergence` (*f\_obs*, *f\_exp*=None, *ddof*=0, *axis*=0, *lambda\_*=None)  
Cressie-Read power divergence statistic and goodness of fit test.

This function tests the null hypothesis that the categorical data has the given frequencies, using the Cressie-Read power divergence statistic.

**Parameters** *f\_obs* : array\_like

Observed frequencies in each category.

*f\_exp* : array\_like, optional

Expected frequencies in each category. By default the categories are assumed to be equally likely.

*ddof* : int, optional

“Delta degrees of freedom”: adjustment to the degrees of freedom for the p-value. The p-value is computed using a chi-squared distribution with  $k - 1 - \text{ddof}$  degrees of freedom, where  $k$  is the number of observed frequencies. The default value of *ddof* is 0.

*axis* : int or None, optional

The axis of the broadcast result of *f\_obs* and *f\_exp* along which to apply the test. If *axis* is None, all values in *f\_obs* are treated as a single data set. Default is 0.

*lambda\_* : float or str, optional

*lambda\_* gives the power in the Cressie-Read power divergence statistic. The default is 1. For convenience, *lambda\_* may be assigned one of the following strings, in which case the corresponding numerical value is used:

String	Value	Description
"pearson"	1	Pearson's chi-squared statistic. In this case, the function is equivalent to <code>`stats.chisquare`</code> .
"log-likelihood"	0	Log-likelihood ratio. Also known as the G-test [R194]_.
"freeman-tukey"	-1/2	Freeman-Tukey statistic.
"mod-log-likelihood"	-1	Modified log-likelihood ratio.
"neyman"	-2	Neyman's statistic.
"cressie-read"	2/3	The power recommended in [R196]_.

**Returns** *statistic* : float or ndarray

The Cressie-Read power divergence test statistic. The value is a float if *axis* is None or if *f\_obs* and *f\_exp* are 1-D.

**pvalue** : float or ndarray

The p-value of the test. The value is a float if *ddof* and the return value *stat* are scalars.

**See also:**

*chisquare*

**Notes**

This test is invalid when the observed or expected frequencies in each category are too small. A typical rule is that all of the observed and expected frequencies should be at least 5.

When *lambda\_* is less than zero, the formula for the statistic involves dividing by *f\_obs*, so a warning or error may be generated if any value in *f\_obs* is 0.

Similarly, a warning or error may be generated if any value in *f\_exp* is zero when *lambda\_*  $\geq$  0.

The default degrees of freedom, *k*-1, are for the case when no parameters of the distribution are estimated. If *p* parameters are estimated by efficient maximum likelihood then the correct degrees of freedom are *k*-1-*p*. If the parameters are estimated in a different way, then the dof can be between *k*-1-*p* and *k*-1. However, it is also possible that the asymptotic distribution is not a chisquare, in which case this test is not appropriate.

This function handles masked arrays. If an element of *f\_obs* or *f\_exp* is masked, then data at that position is ignored, and does not count towards the size of the data set.

New in version 0.13.0.

**References**

[R192], [R193], [R194], [R195], [R196]

**Examples**

(See *chisquare* for more examples.)

When just *f\_obs* is given, it is assumed that the expected frequencies are uniform and given by the mean of the observed frequencies. Here we perform a G-test (i.e. use the log-likelihood ratio statistic):

```
>> from scipy.stats import power_divergence >> power_divergence([16, 18, 16, 14, 12, 12], lambda_='log-likelihood') (2.006573162632538, 0.84823476779463769)
```

The expected frequencies can be given with the *f\_exp* argument:

```
>> power_divergence([16, 18, 16, 14, 12, 12], .. f_exp=[16, 16, 16, 16, 16, 8], .. lambda_='log-likelihood') (3.3281031458963746, 0.6495419288047497)
```

When *f\_obs* is 2-D, by default the test is applied to each column.

```
>> obs = np.array([[16, 18, 16, 14, 12, 12], [32, 24, 16, 28, 20, 24]]).T >> obs.shape (6, 2) >> power_divergence(obs, lambda_='log-likelihood') (array([ 2.00657316, 6.77634498]), array([ 0.84823477, 0.23781225]))
```

By setting *axis*=None, the test is applied to all data in the array, which is equivalent to applying the test to the flattened array.

```
>> power_divergence(obs, axis=None) (23.31034482758621, 0.015975692534127565) >>
power_divergence(obs.ravel()) (23.31034482758621, 0.015975692534127565)
```

*ddof* is the change to make to the default degrees of freedom.

```
>> power_divergence([16, 18, 16, 14, 12, 12], ddof=1) (2.0, 0.73575888234288467)
```

The calculation of the p-values is done by broadcasting the test statistic with *ddof*.

```
>> power_divergence([16, 18, 16, 14, 12, 12], ddof=[0,1,2]) (2.0, array([ 0.84914504, 0.73575888, 0.5724067
]))
```

*f\_obs* and *f\_exp* are also broadcast. In the following, *f\_obs* has shape (6,) and *f\_exp* has shape (2, 6), so the result of broadcasting *f\_obs* and *f\_exp* has shape (2, 6). To compute the desired chi-squared statistics, we must use `axis=1`:

```
>> power_divergence([16, 18, 16, 14, 12, 12], .. f_exp=[[16, 16, 16, 16, 16, 8], .. [8, 20, 20, 16, 12, 12]], ..
axis=1) (array([ 3.5 , 9.25]), array([ 0.62338763, 0.09949846]))
```

`dask.array.stats.skew` (*a*, *axis=0*, *bias=True*, *nan\_policy='propagate'*)

Computes the skewness of a data set.

For normally distributed data, the skewness should be about 0. A skewness value  $> 0$  means that there is more weight in the left tail of the distribution. The function *skewtest* can be used to determine if the skewness value is close enough to 0, statistically speaking.

**Parameters** *a* : ndarray

data

**axis** : int or None, optional

Axis along which skewness is calculated. Default is 0. If None, compute over the whole array *a*.

**bias** : bool, optional

If False, then the calculations are corrected for statistical bias.

**nan\_policy** : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns** *skewness* : ndarray

The skewness of values along an axis, returning 0 where all values are equal.

## References

[R197]

`dask.array.stats.skewtest` (*a*, *axis=0*, *nan\_policy='propagate'*)

Tests whether the skew is different from the normal distribution.

This function tests the null hypothesis that the skewness of the population that the sample was drawn from is the same as that of a corresponding normal distribution.

**Parameters** *a* : array

The data to be tested

**axis** : int or None, optional

Axis along which statistics are calculated. Default is 0. If None, compute over the whole array *a*.

**nan\_policy** : { 'propagate', 'raise', 'omit' }, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns statistic** : float

The computed z-score for this test.

**pvalue** : float

a 2-sided p-value for the hypothesis test

## Notes

The sample size must be at least 8.

`dask.array.stats.kurtosis(a, axis=0, fisher=True, bias=True, nan_policy='propagate')`

Computes the kurtosis (Fisher or Pearson) of a dataset.

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution.

If bias is False then the kurtosis is calculated using k statistics to eliminate bias coming from biased moment estimators

Use *kurtosistest* to see if result is close enough to normal.

**Parameters a** : array

data for which the kurtosis is calculated

**axis** : int or None, optional

Axis along which the kurtosis is calculated. Default is 0. If None, compute over the whole array *a*.

**fisher** : bool, optional

If True, Fisher's definition is used (normal ==> 0.0). If False, Pearson's definition is used (normal ==> 3.0).

**bias** : bool, optional

If False, then the calculations are corrected for statistical bias.

**nan\_policy** : { 'propagate', 'raise', 'omit' }, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns kurtosis** : array

The kurtosis of values along an axis. If all values are equal, return -3 for Fisher's definition and 0 for Pearson's definition.

## References

[R198]

`dask.array.stats.kurtosistest` (*a*, *axis*=0, *nan\_policy*='propagate')

Tests whether a dataset has normal kurtosis

This function tests the null hypothesis that the kurtosis of the population from which the sample was drawn is that of the normal distribution:  $kurtosis = 3(n-1)/(n+1)$ .

**Parameters** *a* : array

array of the sample data

**axis** : int or None, optional

Axis along which to compute test. Default is 0. If None, compute over the whole array *a*.

**nan\_policy** : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns** *statistic* : float

The computed z-score for this test.

**pvalue** : float

The 2-sided p-value for the hypothesis test

## Notes

Valid only for  $n > 20$ . The Z-score is set to 0 for bad entries.

`dask.array.stats.normaltest` (*a*, *axis*=0, *nan\_policy*='propagate')

Tests whether a sample differs from a normal distribution.

This function tests the null hypothesis that a sample comes from a normal distribution. It is based on D'Agostino and Pearson's [R199], [R200] test that combines skew and kurtosis to produce an omnibus test of normality.

**Parameters** *a* : array\_like

The array containing the data to be tested.

**axis** : int or None, optional

Axis along which to compute test. Default is 0. If None, compute over the whole array *a*.

**nan\_policy** : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns** *statistic* : float or array

$s^2 + k^2$ , where *s* is the z-score returned by *skewtest* and *k* is the z-score returned by *kurtosistest*.

**pvalue** : float or array

A 2-sided chi squared probability for the hypothesis test.

## References

[R199], [R200]

`dask.array.stats.f_oneway(*args)`

Performs a 1-way ANOVA.

The one-way ANOVA tests the null hypothesis that two or more groups have the same population mean. The test is applied to samples from two or more groups, possibly with differing sizes.

**Parameters** `sample1, sample2, ..` : array\_like

The sample measurements for each group.

**Returns** `statistic` : float

The computed F-value of the test.

`pvalue` : float

The associated p-value from the F-distribution.

## Notes

The ANOVA test has important assumptions that must be satisfied in order for the associated p-value to be valid.

1. The samples are independent.
2. Each sample is from a normally distributed population.
3. The population standard deviations of the groups are all equal. This property is known as homoscedasticity.

If these assumptions are not true for a given set of data, it may still be possible to use the Kruskal-Wallis H-test (`scipy.stats.kruskal`) although with some loss of power.

The algorithm is from Heiman[2], pp.394-7.

## References

[R201], [R202], [R203]

## Examples

```
>> import scipy.stats as stats
```

[R203] Here are some data on a shell measurement (the length of the anterior adductor muscle scar, standardized by dividing by length) in the mussel *Mytilus trossulus* from five locations: Tillamook, Oregon; Newport, Oregon; Petersburg, Alaska; Magadan, Russia; and Tvarminne, Finland, taken from a much larger data set used in McDonald et al. (1991).

```
>> tillamook = [0.0571, 0.0813, 0.0831, 0.0976, 0.0817, 0.0859, 0.0735, .. 0.0659, 0.0923, 0.0836] >> new-  
port = [0.0873, 0.0662, 0.0672, 0.0819, 0.0749, 0.0649, 0.0835, .. 0.0725] >> petersburg = [0.0974, 0.1352,  
0.0817, 0.1016, 0.0968, 0.1064, 0.105] >> magadan = [0.1033, 0.0915, 0.0781, 0.0685, 0.0677, 0.0697, 0.0764,  
.. 0.0689] >> tvarminne = [0.0703, 0.1026, 0.0956, 0.0973, 0.1039, 0.1045] >> stats.f_oneway(tillamook, new-  
port, petersburg, magadan, tvarminne) (7.1210194716424473, 0.00028122423145345439)
```

`dask.array.stats.moment` (*a*, *moment=1*, *axis=0*, *nan\_policy='propagate'*)

Calculates the *n*th moment about the mean for a sample.

A moment is a specific quantitative measure of the shape of a set of points. It is often used to calculate coefficients of skewness and kurtosis due to its close relationship with them.

**Parameters** *a* : array\_like

data

**moment** : int or array\_like of ints, optional

order of central moment that is returned. Default is 1.

**axis** : int or None, optional

Axis along which the central moment is computed. Default is 0. If None, compute over the whole array *a*.

**nan\_policy** : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

**Returns** *n*-th central moment : ndarray or float

The appropriate moment along the given axis or over all values if axis is None. The denominator for the moment calculation is the number of observations, no degrees of freedom correction is done.

**See also:**

*kurtosis*, *skew*, *describe*

## Notes

The *k*-th central moment of a data sample is:

$$m_k = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^k$$

Where *n* is the number of samples and *x*-bar is the mean. This function uses exponentiation by squares [R204] for efficiency.

## References

[R204]

`dask.array.image.imread` (*filename*, *imread=None*, *preprocess=None*)

Read a stack of images into a dask array

**Parameters** *filename*: string

A globstring like 'myfile.\*.png'

**imread**: function (optional)

Optionally provide custom imread function. Function should expect a filename and produce a numpy array. Defaults to `skimage.io.imread`.

**preprocess**: function (optional)

Optionally provide custom function to preprocess the image. Function should expect a numpy array for a single image.

**Returns** Dask array of all images stacked along the first dimension. All images will be treated as individual chunks

## Examples

```
>>> from dask.array.image import imread
>>> im = imread('2015-**-*.png')
>>> im.shape
(365, 1000, 1000, 3)
```

`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, optional

The dtype of the output array. It is recommended to provide this. If not provided, will be inferred by applying the function to a small set of fake data.

**chunks** : tuple, optional

Chunk shape of resulting blocks if the function does not preserve shape. If not provided, the resulting array is assumed to have the same block structure as the first input array.

**drop\_axis** : number or iterable, optional

Dimensions lost by the function.

**new\_axis** : number or iterable, optional

New dimensions created by the function. Note that these are applied after `drop_axis` (if present).

**token** : string, optional

The key prefix to use for the output array. If not provided, will be determined from the function name.

**name** : string, optional

The key name to use for the output array. Note that this fully specifies the output key name, and must be unique. If not provided, will be determined by a hash of the arguments.

**\*\*kwargs** :

Other keyword arguments to pass to function. Values must be constants (not dask.arrays)

## 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 the function changes shape of the blocks then you must 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 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<func, 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 key name prefix of the resulting task in the graph with the optional `token` keyword argument.

```
>>> x.map_blocks(lambda x: x + 1, token='increment')
dask.array<increment, shape=(100,), dtype=int64, chunksize=(10,)>
```

`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. Many `dask.array` operations are special cases of `atop` including elementwise, broadcasting, reductions, `tensordot`, and `transpose`.

**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

`dtype` : `np.dtype`

Datatype of resulting array.

`concatenate` : bool, keyword only

If true concatenate arrays along dummy indices, else provide lists

`adjust_chunks` : dict

Dictionary mapping index to function to be applied to chunk sizes

`new_axes` : dict, keyword only

New indexes and their dimension lengths

**See also:**

`top`, `contains`

## Examples

2D embarrassingly parallel operation from two arrays, x, and y.

```
>>> z = atop(operator.add, 'ij', x, 'ij', y, 'ij', dtype='f8') # z = x + y
```

Outer product multiplying x by y, two 1-d vectors

```
>>> z = atop(operator.mul, 'ij', x, 'i', y, 'j', dtype='f8')
```

```
z = x.T
```

```
>>> z = atop(np.transpose, 'ji', x, 'ij', dtype=x.dtype)
```

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', dtype='f8')
```

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 iterable of blocks on any array that holds that index. To receive arrays concatenated along contracted dimensions instead pass `concatenate=True`.

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', dtype='f8')
```

Add new single-chunk dimensions with the `new_axes=` keyword, including the length of the new dimension. New dimensions will always be in a single chunk.

```
>>> def f(x):
...     return x[:, None] * np.ones((1, 5))
```

```
>>> z = atop(f, 'az', x, 'a', new_axes={'z': 5}, dtype=x.dtype)
```

If the applied function changes the size of each chunk you can specify this with a `adjust_chunks={...}` dictionary holding a function for each index that modifies the dimension size in that index.

```
>>> def double(x):
...     return np.concatenate([x, x])
```

```
>>> y = atop(double, 'ij', x, 'ij',
...         adjust_chunks={'i': lambda n: 2 * n}, dtype=x.dtype)
```

Include literals by indexing with `None`

```
>>> y = atop(add, 'ij', x, 'ij', 1234, None, dtype=x.dtype)
```

`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).

**See also:**

*atop*

## 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)])}

```

Pass `concatenate=True` to concatenate arrays ahead of time

```

>>> top(f, 'z', 'i', 'x', 'ij', 'y', 'ij', concatenate=True,
...      numblocks={'x': (2, 2), 'y': (2, 2)})
{('z', 0): (f, (concatenate_axes, [('x', 0, 0), ('x', 0, 1)], (1,)),
            (concatenate_axes, [('y', 0, 0), ('y', 0, 1)], (1,))),
 ('z', 1): (f, (concatenate_axes, [('x', 1, 0), ('x', 1, 1)], (1,)),
            (concatenate_axes, [('y', 1, 0), ('y', 1, 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})}

```

Include literals by indexing with `None`

```

>>> top(add, 'z', 'i', 'x', 'i', 100, None, numblocks={'x': (2,)})
{('z', 0): (add, ('x', 0), 100),
 ('z', 1): (add, ('x', 1), 100)}

```

## Array Methods

**class** `dask.array.Array`

Parallel Dask Array

A parallel nd-array comprised of many numpy arrays arranged in a grid.

This constructor is for advanced uses only. For normal use see the `da.from_array` function.

**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

**See also:**

`dask.array.from_array`

**all** (*axis=None, out=None, keepdims=False*)  
Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

**See also:**

`numpy.all` equivalent function

**any** (*axis=None, out=None, keepdims=False*)  
Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

**See also:**

`numpy.any` equivalent function

**argmax** (*axis=None, out=None*)  
Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

**See also:**

`numpy.argmax` equivalent function

**argmin** (*axis=None, out=None*)  
Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.argmin` for detailed documentation.

**See also:**

`numpy.argmin` equivalent function

**astype** (*dtype, \*\*kwargs*)  
Copy of the array, cast to a specified type.

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

Typecode or data-type to which the array is cast.

**casting** : { 'no', 'equiv', 'safe', 'same\_kind', 'unsafe' }, optional

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same\_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

**copy** : bool, optional

By default, `astype` always returns a newly allocated array. If this is set to `False` and the `dtype` requirement is satisfied, the input array is returned instead of a copy.

**choose** (*choices, out=None, mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

**See also:**

`numpy.choose` equivalent function

**clip** (*min=None, max=None, out=None*)

Return an array whose values are limited to `[min, max]`. One of `max` or `min` must be given.

Refer to `numpy.clip` for full documentation.

**See also:**

`numpy.clip` equivalent function

**copy** ()

Copy array. This is a no-op for `dask.arrays`, which are immutable

**cumprod** (*axis, dtype=None, out=None*)

See `da.cumprod` for docstring

**cumsum** (*axis, dtype=None, out=None*)

See `da.cumsum` for docstring

**dot** (*b, out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

**See also:**

`numpy.dot` equivalent function

## Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

**flatten** (*[order]*)

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See also:

`numpy.ravel` equivalent function

`ndarray.flat` a flat iterator on the array.

#### **itemsize**

Length of one array element in bytes

#### **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, optional

The `dtype` of the output array. It is recommended to provide this. If not provided, will be inferred by applying the function to a small set of fake data.

**chunks** : tuple, optional

Chunk shape of resulting blocks if the function does not preserve shape. If not provided, the resulting array is assumed to have the same block structure as the first input array.

**drop\_axis** : number or iterable, optional

Dimensions lost by the function.

**new\_axis** : number or iterable, optional

New dimensions created by the function. Note that these are applied after `drop_axis` (if present).

**token** : string, optional

The key prefix to use for the output array. If not provided, will be determined from the function name.

**name** : string, optional

The key name to use for the output array. Note that this fully specifies the output key name, and must be unique. If not provided, will be determined by a hash of the arguments.

**\*\*kwargs** :

Other keyword arguments to pass to function. Values must be constants (not `dask.arrays`)

## 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 the function changes shape of the blocks then you must 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 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<func, 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 key name prefix of the resulting task in the graph with the optional `token` keyword argument.

```
>>> x.map_blocks(lambda x: x + 1, token='increment')
dask.array<increment, shape=(100,), dtype=int64, chunksize=(10,)>
```

**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, out=None*)

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

**See also:**

**numpy.amax** equivalent function

**mean** (*axis=None, dtype=None, out=None, keepdims=False*)

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

**See also:**

**numpy.mean** equivalent function

**min** (*axis=None, out=None, keepdims=False*)

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

**See also:**

**numpy.amin** equivalent function

**moment** (*order, axis=None, dtype=None, keepdims=False, ddof=0, split\_every=None, out=None*)

Calculate the *n*th centralized moment.

**Parameters** **order** : int

Order of the moment that is returned, must be  $\geq 2$ .

**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  $N - \text{ddof}$ , 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

[R205]

### **nbytes**

Number of bytes in array

### **nonzero** ()

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

#### **See also:**

[numpy.nonzero](#) equivalent function

### **prod** (*axis=None, dtype=None, out=None, keepdims=False*)

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

#### **See also:**

[numpy.prod](#) equivalent function

### **ravel** ([*order* ])

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

#### **See also:**

[numpy.ravel](#) equivalent function

[ndarray.flat](#) a flat iterator on the array.

### **rechunk** (*chunks, threshold=None, block\_size\_limit=None*)

See *da.rechunk* for docstring

### **repeat** (*repeats, axis=None*)

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

#### **See also:**

[numpy.repeat](#) equivalent function

### **reshape** (*shape, order='C'*)

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

#### **See also:**

[numpy.reshape](#) equivalent function

**round** (*decimals=0, out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

**See also:**

`numpy.around` equivalent function

**size**

Number of elements in array

**squeeze** (*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

**See also:**

`numpy.squeeze` equivalent function

**std** (*axis=None, dtype=None, out=None, ddof=0, keepdims=False*)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

**See also:**

`numpy.std` equivalent function

**store** (*sources, targets, lock=True, regions=None, compute=True, return\_stored=False, \*\*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.

**regions:** tuple of slices or iterable of tuple of slices

Each region tuple in `regions` should be such that `target[region].shape = source.shape` for the corresponding source and target in `sources` and `targets`, respectively.

**compute:** boolean, optional

If true compute immediately, return `dask.delayed.Delayed` otherwise

**return\_stored:** boolean, optional

Optionally return the stored result (default `False`).

## 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, out=None, keepdims=False*)  
Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

**See also:**

[`numpy.sum`](#) equivalent function

**swapaxes** (*axis1, axis2*)  
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

**See also:**

[`numpy.swapaxes`](#) equivalent function

**to\_dask\_dataframe** (*columns=None*)  
Convert dask Array to dask Dataframe

**Parameters** **columns:** list or string

list of column names if DataFrame, single string if Series

**See also:**

[`dask.dataframe.from\_dask\_array`](#)

**to\_delayed** ()  
Convert Array into dask Delayed objects  
Returns an array of values, one value per chunk.

**See also:**

[`dask.array.from\_delayed`](#)

**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.

See `da.topk` for docstring

**transpose** (*\*axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an *n*-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters** `axes` : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

**Returns** `out` : ndarray

View of *a*, with axes suitably permuted.

**See also:**

`ndarray.T` Array property returning the array transposed.

**Examples**

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

**var** (*axis=None, dtype=None, out=None, ddof=0, keepdims=False*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

**See also:**

`numpy.var` equivalent function

**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.**

**vindex**

Vectorized indexing with broadcasting.

This is equivalent to numpy's advanced indexing, using arrays that are broadcast against each other. This allows for pointwise indexing:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x = from_array(x, chunks=2)
>>> x.vindex[[0, 1, 2], [0, 1, 2]].compute()
array([1, 5, 9])
```

Mixed basic/advanced indexing with slices/arrays is also supported. The order of dimensions in the result follows those proposed for `ndarray.vindex [1]`: the subspace spanned by arrays is followed by all slices.

Note: `vindex` provides more general functionality than standard indexing, but it also has fewer optimizations and can be significantly slower.

[1]: <https://github.com/numpy/numpy/pull/6256>

**vnorm** (*ord=None*, *axis=None*, *keepdims=False*, *split\_every=None*, *out=None*)

Vector norm

Other topics

## 4.6.6 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.

## 4.6.7 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 = [da.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 = [da.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)
```

### 4.6.8 Overlapping Blocks with Ghost Cells

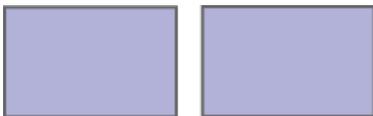
Some array operations require communication of borders between neighboring blocks. Example operations include the following:

- 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*

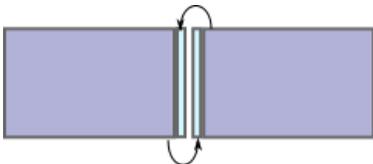
Dask array supports these operations by creating a new array where each block is slightly expanded by the borders of its neighbors. This costs an excess copy and the communication of many small chunks but allows localized functions to evaluate in an embarrassing manner. We call this process *ghosting*.

#### Ghosting

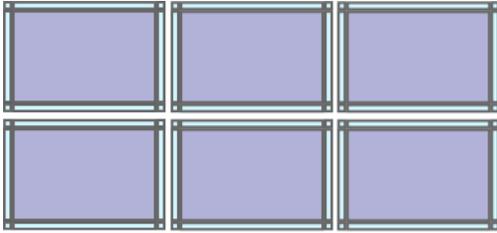
Consider two neighboring blocks in a Dask array.



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))

>>> y = da.ghost.trim_internal(x, {0: 2, 1: 1})
>>> y.chunks
((6, 6, 6, 6), (8, 8, 8, 8))
```

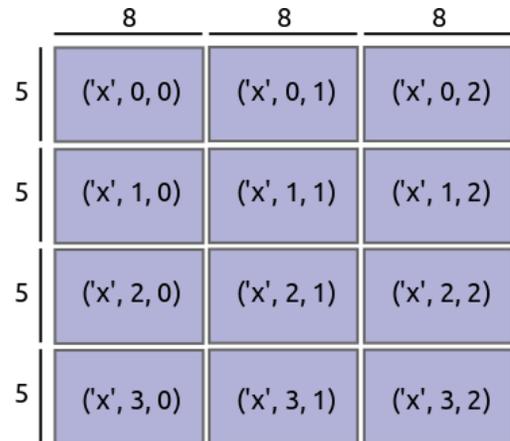
## Full Workflow

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})
```

## 4.6.9 Internal Design

### Overview



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 graph with a special set of keys designating blocks such as ('x', 0, 0), ('x', 0, 1), ... (See [Dask graph documentation](#) for more details.)
- 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 graph refer to this array, like 'x'
- A NumPy dtype

### Example

```
>>> import dask.array as da
>>> x = da.arange(0, 15, chunks=(5,))

>>> x.name
'arange-539766a'

>>> x.dask # somewhat simplified
{('arange-539766a', 0): (np.arange, 0, 5),
 ('arange-539766a', 1): (np.arange, 5, 10),
 ('arange-539766a', 2): (np.arange, 10, 15)}

>>> x.chunks
((5, 5, 5),)

>>> x.dtype
dtype('int64')
```

### 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 graph 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 graphs, update dtypes, and track chunk 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)
```

## 4.6.10 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.*

## 4.6.11 Sparse Arrays

By swapping out in-memory numpy arrays with in-memory sparse arrays we can reuse the blocked algorithms of `Dask.array` to achieve parallel and distributed sparse arrays.

The blocked algorithms in `Dask.array` normally parallelize around in-memory numpy arrays. However, if another in-memory array library supports the NumPy interface then it too can take advantage of `dask.array`'s parallel algorithms.

In particular the `sparse` array library satisfies a subset of the NumPy API and works well with, and is tested against, `Dask.array`.

## Example

Say we have a `dask.array` with mostly zeros

```
x = da.random.random((100000, 100000), chunks=(1000, 1000))
x[x < 0.95] = 0
```

We can convert each of these chunks of NumPy arrays into a `sparse.COO` array.

```
import sparse
s = x.map_blocks(sparse.COO)
```

Now our array is composed not of many NumPy arrays, but rather of many sparse arrays. Semantically this does not change anything. Operations that work will work identically (assuming that the behavior of `numpy` and `sparse` are identical) but performance characteristics and storage costs may change significantly

```
>>> s.sum(axis=0)[:100].compute()
<COO: shape=(100,), dtype=float64, nnz=100>

>>> _.todense()
array([[ 4803.06859272,  4913.94964525,  4877.13266438,  4860.7470773 ,
         4938.94446802,  4849.51326473,  4858.83977856,  4847.81468485,
         ... ]])
```

## Requirements

Any in-memory library that copies the NumPy ndarray interface should work here. The `sparse` library is a minimal example. In particular an in-memory library should implement at least the following operations:

1. Simple slicing with slices, lists, and elements (for slicing, rechunking, reshaping, etc).
2. A `concatenate` function matching the interface of `np.concatenate`. This must be registered in `dask.array.core.concatenate_lookup`.
3. All ufuncs must support the full ufunc interface, including `dtype=` and `out=` parameters (even if they don't function properly)
4. All reductions must support the full `axis=` and `keepdims=` keywords and behave like `numpy` in this respect
5. The array class should follow the `__array_priority__` protocol and be prepared to respond to other arrays of lower priority.
6. If `dot` support is desired, a `tensordot` function matching the interface of `np.tensordot` should be registered in `dask.array.core.tensordot_lookup`.

The implementation of other operations like `reshape`, `transpose`, etc. should follow standard NumPy conventions regarding shape and `dtype`. Not implementing these is fine; the parallel `dask.array` will err at runtime if these operations are attempted.

## Mixed Arrays

`Dask.array` supports mixing different kinds of in-memory arrays. This relies on the in-memory arrays knowing how to interact with each other when necessary. When two arrays interact the functions from the array with the highest `__array_priority__` will take precedence (for example for `concatenate`, `tensordot`, etc.).

## 4.6.12 Stats

Dask Array implements a subset of the `scipy.stats` package.

### Statistical Functions

You can calculate various measures of an array including skewness, kurtosis, and arbitrary moments.

```
>>> from dask.array import stats
>>> x = da.random.beta(1, 1, size=(1000,), chunks=10)
>>> k, s, m = [stats.kurtosis(x), stats.skew(x), stats.moment(x, 5)]
>>> dask.compute(k, s, m)
(1.7612340817172787, -0.064073498030693302, -0.00054523780628304799)
```

### Statistical Tests

You can perform basic statistical tests on dask arrays. Each of these tests return a `dask.delayed` wrapping one of the `scipy` `namedtuple` results.

```
>>> a = da.random.uniform(size=(50,), chunks=(25,))
>>> b = a + da.random.uniform(low=-0.15, high=0.15, size=(50,), chunks=(25,))
>>> result = ttest_rel(a, b)
>>> result.compute()
Ttest_relResult(statistic=-1.5102104380013242, pvalue=0.13741197274874514)
```

## 4.7 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.

### 4.7.1 Overview

`Dask.Bag` implements a operations like `map`, `filter`, `fold`, and `groupby` on collections of Python objects. It does this in parallel and in small memory using Python iterators. It is similar to a parallel version of `PyToolz` or a Pythonic version of the `PySpark RDD`.

#### Design

Dask bags coordinate many Python lists or Iterators, each of which forms a partition of a larger 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. Parallel: data is split up, allowing multiple cores or machines to execute in parallel.
2. Iterating: data processes lazily, allowing smooth execution of larger-than-memory data, even on a single machine within a single partition

## 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 `groupby`, require substantial inter-worker communication. On a single machine, dask uses `partd` to perform efficient, parallel, spill-to-disk shuffles. When working in a cluster, dask uses a task based shuffle.

These 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.

## 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. Bags are immutable and so you can not change individual elements
3. Bag operations tend to be slower than array/dataframe computations in the same way that standard Python containers tend to be slower than NumPy arrays and Pandas dataframes.
4. `Bag.groupby` is slow. You should try to use `Bag.foldby` if possible. Using `Bag.foldby` requires more thought.

## 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 *i*th element.

## 4.7.2 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 iterable:

```
>>> 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.txt')
>>> b = db.read_text(['myfile.1.txt', 'myfile.2.txt', ...])
>>> b = db.read_text('myfile.*.txt')
```

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.*.txt.gz')
```

The resulting items in the bag are strings. If you have encoded data like line-delimited JSON then you may want to map a decoding or load function across the bag:

```
>>> import json
>>> b = db.read_text('myfile.*.json').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').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.

### 4.7.3 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, get=None, storage_options=None)
```

Write dask 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` (*meta=None, columns=None*)

Create Dask Dataframe from a Dask Bag.

Bag should contain tuples, dict records, or scalars.

Index will not be particularly meaningful. Use `reindex` afterwards if necessary.

**Parameters** `meta` : `pd.DataFrame`, dict, iterable, optional

An empty `pd.DataFrame` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. If not provided or a list, a single element from the first partition will be computed, triggering a potentially expensive call to `compute`. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**columns** : sequence, optional

Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns). Note that if `meta` is provided, column names will be taken from there and this parameter is invalid.

## 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 list of dask Delayed.  
 Returns list of Delayed, one per partition.

## 4.7.4 API

Top level user functions:

<code>Bag(dsk, name, npartitions)</code>	Parallel collection of Python objects
<code>Bag.all([split_every])</code>	Are all elements truthy?
<code>Bag.any([split_every])</code>	Are any of the elements truthy?
<code>Bag.compute(**kwargs)</code>	Compute this dask collection
<code>Bag.count([split_every])</code>	Count the number of elements.
<code>Bag.distinct()</code>	Distinct elements of collection
<code>Bag.filter(predicate)</code>	Filter elements in collection by a predicate function.
<code>Bag.flatten()</code>	Concatenate nested lists into one long list.
<code>Bag.fold(binop[, combine, initial, split_every])</code>	Parallelizable reduction
<code>Bag.foldby(key, binop[, initial, combine, ...])</code>	Combined reduction and groupby.
<code>Bag.frequencies([split_every])</code>	Count number of occurrences of each distinct element.
<code>Bag.groupby(grouper[, method, npartitions, ...])</code>	Group collection by key function
<code>Bag.join(other, on_self[, on_other])</code>	Joins collection with another collection.
<code>Bag.map(func, *args, **kwargs)</code>	Apply a function elementwise across one or more bags.
<code>Bag.map_partitions(func, *args, **kwargs)</code>	Apply a function to every partition across one or more bags.
<code>Bag.max([split_every])</code>	Maximum element
<code>Bag.mean()</code>	Arithmetic mean
<code>Bag.min([split_every])</code>	Minimum element
<code>Bag.pluck(key[, default])</code>	Select item from all tuples/dicts in collection.
<code>Bag.product(other)</code>	Cartesian product between two bags.
<code>Bag.reduction(perpartition, aggregate[, ...])</code>	Reduce collection with reduction operators.
<code>Bag.random_sample(prob[, random_state])</code>	Return elements from bag with probability of <code>prob</code> .
<code>Bag.remove(predicate)</code>	Remove elements in collection that match predicate.
<code>Bag.repartition(npartitions)</code>	Coalesce bag into fewer partitions.
<code>Bag.starmap(func, **kwargs)</code>	Apply a function using argument tuples from the given bag.
<code>Bag.std([ddof])</code>	Standard deviation
<code>Bag.sum([split_every])</code>	Sum all elements
<code>Bag.take(k[, npartitions, compute, warn])</code>	Take the first <code>k</code> elements.
<code>Bag.to_dataframe([meta, columns])</code>	Create Dask Dataframe from a Dask Bag.
<code>Bag.to_delayed()</code>	Convert bag to list of dask Delayed.
<code>Bag.to_textfiles(b, path[, name_function, ...])</code>	Write dask Bag to disk, one filename per partition, one line per element.
<code>Bag.topk(k[, key, split_every])</code>	<code>K</code> largest elements in collection
<code>Bag.var([ddof])</code>	Variance
<code>Bag.visualize([filename, format, optimize_graph])</code>	Render the computation of this object's task graph using <code>graphviz</code> .

## Create Bags

<code>from_sequence(seq[, partition_size, npartitions])</code>	Create a dask Bag from Python sequence.
--	---

Continued on next page

Table 4.12 – continued from previous page

<code>from_delayed(values)</code>	Create bag from many dask Delayed objects.
<code>read_text(urlpath[, blocksize, compression, ...])</code>	Read lines from text files
<code>from_url(urls)</code>	Create a dask Bag from a url.
<code>range(n, npartitions)</code>	Numbers from zero to n

## Top-level functions

<code>concat(bags)</code>	Concatenate many bags together, unioning all elements.
<code>map(func, *args, **kwargs)</code>	Apply a function elementwise across one or more bags.
<code>map_partitions(func, *args, **kwargs)</code>	Apply a function to every partition across one or more bags.
<code>zip(*bags)</code>	Partition-wise bag zip

## Turn Bags into other things

<code>Bag.to_textfiles(b, path[, name_function, ...])</code>	Write dask Bag to disk, one filename per partition, one line per element.
<code>Bag.to_dataframe([meta, columns])</code>	Create Dask Dataframe from a Dask Bag.
<code>Bag.to_delayed()</code>	Convert bag to list of dask Delayed.

## 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.

This assumes that the bag is ordered. While this is typically the case not all Dask.bag functions preserve this property.

### 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, initial=-1)
[-1, 0, 2, 5, 9, 14]
```

**all** (*split\_every=None*)

Are all elements truthy?

**any** (*split\_every=None*)

Are any of the elements truthy?

**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]
```

**flatten** ()

Concatenate nested lists into one long list.

```
>>> b = from_sequence([[1], [2, 3]])
>>> list(b)
[[1], [2, 3]]
```

```
>>> list(b.flatten())
[1, 2, 3]
```

**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\_\_'*, *split\_every=None*)

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.

### split\_every

Group partitions into groups of this size while performing reduction. Defaults to 8.

```
>>> 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 (bytes)

**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*)

Joins 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, \*args, \*\*kwargs*)

Apply a function elementwise across one or more bags.

Note that all Bag arguments must be partitioned identically.

**Parameters func** : callable

**\*args, \*\*kwargs** : Bag, Item, or object

Extra arguments and keyword arguments to pass to `func` *after* the calling bag instance. Non-Bag args/kwags are broadcasted across all calls to `func`.

## Notes

For calls with multiple *Bag* arguments, corresponding partitions should have the same length; if they do not, the call will error at compute time.

## Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5), npartitions=2)
>>> b2 = db.from_sequence(range(5, 10), npartitions=2)
```

Apply a function to all elements in a bag:

```
>>> b.map(lambda x: x + 1).compute()
[1, 2, 3, 4, 5]
```

Apply a function with arguments from multiple bags:

```
>>> from operator import add
>>> b.map(add, b2).compute()
[5, 7, 9, 11, 13]
```

Non-bag arguments are broadcast across all calls to the mapped function:

```
>>> b.map(add, 1).compute()
[1, 2, 3, 4, 5]
```

Keyword arguments are also supported, and have the same semantics as regular arguments:

```
>>> def myadd(x, y=0):
...     return x + y
>>> b.map(myadd, y=b2).compute()
[5, 7, 9, 11, 13]
>>> b.map(myadd, y=1).compute()
[1, 2, 3, 4, 5]
```

Both arguments and keyword arguments can also be instances of `dask.bag.Item`. Here we'll add the max value in the bag to each element:

```
>>> b.map(myadd, b.max()).compute()
[4, 5, 6, 7, 8]
```

**map\_partitions** (*func*, *\*args*, *\*\*kwargs*)

Apply a function to every partition across one or more bags.

Note that all Bag arguments must be partitioned identically.

**Parameters** *func* : callable

*\*args*, *\*\*kwargs* : Bag, Item, Delayed, or object

Arguments and keyword arguments to pass to *func*. Partitions from this bag will be the first argument, and these will be passed *after*.

## 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** (*split\_every=None*)  
Maximum element

**mean** ()  
Arithmetic mean

**min** (*split\_every=None*)  
Minimum element

**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*.

**Parameters** *prob* : float

A float between 0 and 1, representing the probability that each element will be returned.

**random\_state** : int or random.Random, optional

If an integer, will be used to seed a new `random.Random` object. If provided, results in deterministic sampling.

## Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.random_sample(0.5, 42))
[1, 4]
>>> list(b.random_sample(0.5, 42))
[1, 4]
```

**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
```

**starmap** (*func*, *\*\*kwargs*)

Apply a function using argument tuples from the given bag.

This is similar to `itertools.starmap`, except it also accepts keyword arguments. In pseudocode, this could be written as:

```
>>> def starmap(func, bag, **kwargs):
...     return (func(*args, **kwargs) for args in bag)
```

**Parameters** `func` : callable

**`**kwargs`** : Item, Delayed, or object, optional

Extra keyword arguments to pass to `func`. These can either be normal objects, `dask.bag.Item`, or `dask.delayed.Delayed`.

## Examples

```
>>> import dask.bag as db
>>> data = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
>>> b = db.from_sequence(data, npartitions=2)
```

Apply a function to each argument tuple:

```
>>> from operator import add
>>> b.starmap(add).compute()
[3, 7, 11, 15, 19]
```

Apply a function to each argument tuple, with additional keyword arguments:

```
>>> def myadd(x, y, z=0):
...     return x + y + z
>>> b.starmap(myadd, z=10).compute()
[13, 17, 21, 25, 29]
```

Keyword arguments can also be instances of `dask.bag.Item` or `dask.delayed.Delayed`:

```
>>> max_second = b.pluck(1).max()
>>> max_second.compute()
10
>>> b.starmap(myadd, z=max_second).compute()
[13, 17, 21, 25, 29]
```

**std** (*ddof=0*)

Standard deviation

**str**

String processing functions

## Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence(['Alice Smith', 'Bob Jones', 'Charlie Smith'])
>>> list(b.str.lower())
['alice smith', 'bob jones', 'charlie smith']
```

```
>>> list(b.str.match('*Smith'))
['Alice Smith', 'Charlie Smith']
```

```
>>> list(b.str.split(' '))
[['Alice', 'Smith'], ['Bob', 'Jones'], ['Charlie', 'Smith']]
```

**sum** (*split\_every=None*)  
Sum all elements

**take** (*k*, *npartitions=1*, *compute=True*, *warn=True*)  
Take the first *k* elements.

**Parameters** **k** : int

The number of elements to return

**npartitions** : int, optional

Elements are only taken from the first *npartitions*, with a default of 1. If there are fewer than *k* rows in the first *npartitions* a warning will be raised and any found rows returned. Pass -1 to use all partitions.

**compute** : bool, optional

Whether to compute the result, default is True.

**warn** : bool, optional

Whether to warn if the number of elements returned is less than requested, default is True.

```
>>> b = from_sequence(range(10))
```

```
>>> b.take(3) # doctest: +SKIP
```

```
(0, 1, 2)
```

**to\_dataframe** (*meta=None*, *columns=None*)  
Create Dask Dataframe from a Dask Bag.

Bag should contain tuples, dict records, or scalars.

Index will not be particularly meaningful. Use `reindex` afterwards if necessary.

**Parameters** **meta** : `pd.DataFrame`, dict, iterable, optional

An empty `pd.DataFrame` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. If not provided or a list, a single element from the first partition will be computed, triggering a potentially expensive call to `compute`. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**columns** : sequence, optional

Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns). Note that if `meta` is provided, column names will be taken from there and this parameter is invalid.

## 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 list of dask Delayed.

Returns list of Delayed, one per partition.

### `to_textfiles(b, path, name_function=None, compression='infer', encoding='utf-8', compute=True, get=None, storage_options=None)`

Write dask 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 a dask Bag 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 text files

## Examples

```
>>> b = from_sequence(['Alice', 'Bob', 'Chuck'], partition_size=2)
```

`dask.bag.from_delayed(values)`

Create bag from many dask Delayed objects.

These objects will become the partitions of the resulting Bag. They should evaluate to a `list` or some other concrete sequence.

**Parameters values: list of delayed values**

An iterable of dask Delayed objects. Each evaluating to a list.

**Returns** Bag

**See also:**

`dask.delayed`

## Examples

```
>>> x, y, z = [delayed(load_sequence_from_file)(fn)
...           for fn in filenames]
>>> 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 (in bytes) 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\_url**(*urls*)

Create a dask Bag from a url.

**Examples**

```
>>> a = from_url('http://raw.githubusercontent.com/dask/dask/master/README.rst')
>>> a.npartitions
1
```

```
>>> a.take(8)
(b'Dask\n',
 b'====\n',
 b'\n',
 b'|Build Status| |Coverage| |Doc Status| |Gitter| |Version Status|\n',
 b'\n',
 b'Dask is a flexible parallel computing library for analytics. See\n',
 b'documentation_ for more information.\n',
 b'\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.map_partitions` (*func*, *\*args*, *\*\*kwargs*)

Apply a function to every partition across one or more bags.

Note that all Bag arguments must be partitioned identically.

**Parameters** `func` : callable

`*args`, `**kwargs` : Bag, Item, Delayed, or object

Arguments and keyword arguments to pass to `func`.

## 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.

`dask.bag.map` (*func*, *\*args*, *\*\*kwargs*)

Apply a function elementwise across one or more bags.

Note that all Bag arguments must be partitioned identically.

**Parameters** `func` : callable

`*args, **kwargs` : Bag, Item, Delayed, or object

Arguments and keyword arguments to pass to `func`. Non-Bag args/kwags are broadcasted across all calls to `func`.

## Notes

For calls with multiple *Bag* arguments, corresponding partitions should have the same length; if they do not, the call will error at compute time.

## Examples

```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5), npartitions=2)
>>> b2 = db.from_sequence(range(5, 10), npartitions=2)
```

Apply a function to all elements in a bag:

```
>>> db.map(lambda x: x + 1, b).compute()
[1, 2, 3, 4, 5]
```

Apply a function with arguments from multiple bags:

```
>>> from operator import add
>>> db.map(add, b, b2).compute()
[5, 7, 9, 11, 13]
```

Non-bag arguments are broadcast across all calls to the mapped function:

```
>>> db.map(add, b, 1).compute()
[1, 2, 3, 4, 5]
```

Keyword arguments are also supported, and have the same semantics as regular arguments:

```
>>> def myadd(x, y=0):
...     return x + y
>>> db.map(myadd, b, y=b2).compute()
[5, 7, 9, 11, 13]
>>> db.map(myadd, b, y=1).compute()
[1, 2, 3, 4, 5]
```

Both arguments and keyword arguments can also be instances of `dask.bag.Item` or `dask.delayed.Delayed`. Here we'll add the max value in the bag to each element:

```
>>> db.map(myadd, b, b.max()).compute()
[4, 5, 6, 7, 8]
```

`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(fizzbuzz)
[(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 the following:

```
>>> list(fizzbuzz)
[(0, 0), (3, None), (None, 5), (6, None), (None, 10), (9, None),
(12, None), (15, 15), (18, None), (None, 20), (None, 25), (None, 30)]
```

## 4.8 DataFrame

A Dask DataFrame is a large parallel dataframe composed of many smaller Pandas dataframes, split along the index. These pandas dataframes may live on disk for larger-than-memory computing on a single machine, or on many different machines in a cluster.

Dask.dataframe implements a commonly used subset of the Pandas interface including elementwise operations, reductions, grouping operations, joins, timeseries algorithms, and more. It copies the Pandas interface for these operations exactly and so should be very familiar to Pandas users. Because Dask.dataframe operations merely coordinate Pandas operations they usually exhibit similar performance characteristics as are found in Pandas.

### 4.8.1 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 or on many dataframes spread across a cluster. One operation on a dask.dataframe triggers many operations on the constituent Pandas dataframes.

### Design

Dask dataframes coordinate many Pandas DataFrames/Series arranged along the index. Dask.dataframe is partitioned *row-wise*, grouping rows by index value for efficiency. These Pandas objects may live on disk or on other machines.

## Common Uses and Anti-Uses

Dask.dataframe is particularly useful in the following situations:

- Manipulating large datasets on a single machine, even when those datasets don't fit comfortably into memory.
- Fast computation on large workstation machines by parallelizing many Pandas calls across many cores.
- Distributed computing of very large tables stored in the Hadoop File System (HDFS), S3, or other parallel file systems.
- Parallel groupby, join, or time series computations

However in the following situations Dask.dataframe may not be the best choice:

- If your dataset fits comfortably into RAM on your laptop then you may be better off just using [Pandas](#). There may be simpler ways to improve performance than through parallelism.
- If your dataset doesn't fit neatly into the Pandas tabular model then you might find more use in [dask.bag](#) or [dask.array](#)
- If you need functions that are not implemented in dask.dataframe then you might want to look at [dask.delayed](#) which offers more flexibility.
- If you need a proper database with all that databases offer you might prefer something like [Postgres](#)

## 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()`
- groupby-apply on index: `df.groupby(['idx', 'x']).apply(myfunc)`, where `idx` is the index level name
- 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)` or `dd.merge(df1, df2, on=['idx', 'x'])` where `idx` is the index name for both `df1` and `df2`
- 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 not on index (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`

### 4.8.2 Create and Store Dask DataFrames

Dask can create dataframes from various data storage formats like CSV, HDF, Apache Parquet, and others. For most formats this data can live on various storage systems including local disk, network file systems (NFS), the Hadoop File System (HDFS), and Amazon's S3 (excepting HDF, which is only available on POSIX like file systems).

See the [Overview section](#) for an in depth discussion of `dask.dataframe` scope, use, limitations.

## API

The following functions provide access to convert between Dask Dataframes, file formats, and other Dask or Python collections.

File Formats:

<code>read_csv(urlpath[, blocksize, collection, ...])</code>	Read CSV files into a Dask.DataFrame
<code>read_parquet(path[, columns, filters, ...])</code>	Read ParquetFile into a Dask DataFrame
<code>read_hdf(pattern, key[, start, stop, ...])</code>	Read HDF files into a Dask DataFrame
<code>read_sql_table(table, uri, index_col[, ...])</code>	Create dataframe from an SQL table.
<code>from_bcolz(x[, chunksize, categorize, ...])</code>	Read BColz CTable into a Dask Dataframe
<code>from_array(x[, chunksize, columns])</code>	Read any slicable array into a Dask Dataframe
<code>to_csv(df, filename[, name_function, ...])</code>	Store Dask DataFrame to CSV files
<code>to_parquet(df, path[, engine, compression, ...])</code>	Store Dask.dataframe to Parquet files
<code>to_hdf(df, path, key[, mode, append, get, ...])</code>	Store Dask Dataframe to Hierarchical Data Format (HDF) files

Dask Collections:

<code>from_delayed(dfs[, meta, divisions, prefix])</code>	Create Dask DataFrame from many Dask Delayed objects
<code>from_dask_array(x[, columns])</code>	Create a Dask DataFrame from a Dask Array.
<code>dask.bag.core.Bag.to_dataframe([meta, columns])</code>	Create Dask Dataframe from a Dask Bag.
<code>to_delayed(df)</code>	Create Dask Delayed objects from a Dask Dataframe
<code>to_records(df)</code>	Create Dask Array from a Dask Dataframe
<code>to_bag(df[, index])</code>	Create Dask Bag from a Dask DataFrame

Pandas:

---

<code>from_pandas(data[, npartitions, chunksize, ...])</code>	Construct a Dask DataFrame from a Pandas DataFrame
---	--

---

## Locations

For text, CSV, and Apache Parquet formats data can come from local disk, from the Hadoop File System, from S3FS, or others, by prepending the filenames with a protocol.

```
>>> df = dd.read_csv('my-data-*.csv')
>>> df = dd.read_csv('hdfs:///path/to/my-data-*.csv')
>>> df = dd.read_csv('s3://bucket-name/my-data-*.csv')
```

For remote systems like HDFS or S3 credentials may be an issue. Usually these are handled by configuration files on disk (such as a `.boto` file for S3) but in some cases you may want to pass storage-specific options through to the storage backend. You can do this with the `storage_options=` keyword.

```
>>> df = dd.read_csv('s3://bucket-name/my-data-*.csv',
...                  storage_options={'anon': True})
```

## Dask Delayed

For more complex situations not covered by the functions above you may want to use `dask.delayed`, which lets you construct `Dask.dataframes` out of arbitrary Python function calls that load dataframes. This can allow you to handle new formats easily, or bake in particular logic around loading data if, for example, your data is stored with some special

See [documentation on using dask.delayed with collections](#) or an [example notebook](#) showing how to create a Dask DataFrame from a nested directory structure of Feather files (as a stand in for any custom file format).

`Dask.delayed` is particularly useful when simple `map` operations aren't sufficient to capture the complexity of your data layout.

## From Raw Dask Graphs

This section is mainly for developers wishing to extend `dask.dataframe`. It discusses internal API not normally needed by users. Everything below can be done just as effectively with `dask.delayed` described just above. 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)
```

### 4.8.3 API

#### Dataframe

<code>DataFrame(dsk, name, meta, divisions)</code>	Parallel Pandas DataFrame
<code>DataFrame.add(other[, axis, level, fill_value])</code>	Addition of dataframe and other, element-wise (binary operator <i>add</i> ).
<code>DataFrame.append(other)</code>	Append rows of <i>other</i> to the end of this frame, returning a new object.
<code>DataFrame.apply(func[, axis, args, meta])</code>	Parallel version of <code>pandas.DataFrame.apply</code>
<code>DataFrame.assign(**kwargs)</code>	Assign new columns to a DataFrame, returning a new object (a copy) with all the original columns in addition to the new ones.
<code>DataFrame.astype(dtype)</code>	Cast a pandas object to a specified dtype <code>dtype</code> .
<code>DataFrame.categorize(df[, columns, index, ...])</code>	Convert columns of the DataFrame to category dtype.
<code>DataFrame.columns</code>	
<code>DataFrame.compute(**kwargs)</code>	Compute this dask collection
<code>DataFrame.corr([method, min_periods, ...])</code>	Compute pairwise correlation of columns, excluding NA/null values
<code>DataFrame.count([axis, split_every])</code>	Return Series with number of non-NA/null observations over requested axis.
<code>DataFrame.cov([min_periods, split_every])</code>	Compute pairwise covariance of columns, excluding NA/null values
<code>DataFrame.cummax([axis, skipna])</code>	Return cumulative max over requested axis.
<code>DataFrame.cummin([axis, skipna])</code>	Return cumulative minimum over requested axis.
<code>DataFrame.cumprod([axis, skipna])</code>	Return cumulative product over requested axis.
<code>DataFrame.cumsum([axis, skipna])</code>	Return cumulative sum over requested axis.
<code>DataFrame.describe([split_every])</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>DataFrame.div(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>DataFrame.drop(labels[, axis, errors])</code>	Return new object with labels in requested axis removed.
<code>DataFrame.drop_duplicates([split_every, ...])</code>	Return DataFrame with duplicate rows removed, optionally only
<code>DataFrame.dropna([how, subset])</code>	Return object with labels on given axis omitted where alternately any
<code>DataFrame.dtypes</code>	Return data types

Continued on next page

Table 4.18 – continued from previous page

<code>DataFrame.fillna([value, method, limit, axis])</code>	Fill NA/NaN values using the specified method
<code>DataFrame.floordiv(other[, axis, level, ...])</code>	Integer division of dataframe and other, element-wise (binary operator <i>floordiv</i> ).
<code>DataFrame.get_partition(n)</code>	Get a dask DataFrame/Series representing the <i>n</i> th partition.
<code>DataFrame.groupby([by])</code>	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
<code>DataFrame.head([n, npartitions, compute])</code>	First <i>n</i> rows of the dataset
<code>DataFrame.index</code>	Return dask Index instance
<code>DataFrame.iterrows()</code>	Iterate over DataFrame rows as (index, Series) pairs.
<code>DataFrame.itertuples()</code>	Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.
<code>DataFrame.join(other[, on, how, lsuffix, ...])</code>	Join columns with other DataFrame either on index or on a key column.
<code>DataFrame.known_divisions</code>	Whether divisions are already known
<code>DataFrame.loc</code>	Purely label-location based indexer for selection by label.
<code>DataFrame.map_partitions(func, *args, **kwargs)</code>	Apply Python function on each DataFrame partition.
<code>DataFrame.mask(cond[, other])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is False and otherwise are from <i>other</i> .
<code>DataFrame.max([axis, skipna, split_every])</code>	This method returns the maximum of the values in the object.
<code>DataFrame.mean([axis, skipna, split_every])</code>	Return the mean of the values for the requested axis
<code>DataFrame.merge(right[, how, on, left_on, ...])</code>	Merge DataFrame objects by performing a database-style join operation by columns or indexes.
<code>DataFrame.min([axis, skipna, split_every])</code>	This method returns the minimum of the values in the object.
<code>DataFrame.mod(other[, axis, level, fill_value])</code>	Modulo of dataframe and other, element-wise (binary operator <i>mod</i> ).
<code>DataFrame.mul(other[, axis, level, fill_value])</code>	Multiplication of dataframe and other, element-wise (binary operator <i>mul</i> ).
<code>DataFrame.ndim</code>	Return dimensionality
<code>DataFrame.nlargest([n, columns, split_every])</code>	Get the rows of a DataFrame sorted by the <i>n</i> largest values of <i>columns</i> .
<code>DataFrame.npartitions</code>	Return number of partitions
<code>DataFrame.pow(other[, axis, level, fill_value])</code>	Exponential power of dataframe and other, element-wise (binary operator <i>pow</i> ).
<code>DataFrame.quantile([q, axis])</code>	Approximate row-wise and precise column-wise quantiles of DataFrame
<code>DataFrame.query(expr, **kwargs)</code>	Filter dataframe with complex expression
<code>DataFrame.radd(other[, axis, level, fill_value])</code>	Addition of dataframe and other, element-wise (binary operator <i>radd</i> ).
<code>DataFrame.random_split(frac[, random_state])</code>	Pseudorandomly split dataframe into different pieces row-wise
<code>DataFrame.rdiv(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <i>rtruediv</i> ).
<code>DataFrame.rename([index, columns])</code>	Alter axes labels.
<code>DataFrame.repartition([divisions, ...])</code>	Repartition dataframe along new divisions
<code>DataFrame.reset_index([drop])</code>	Reset the index to the default index.

Continued on next page

Table 4.18 – continued from previous page

<code>DataFrame.rfloordiv</code> (other[, axis, level, ...])	Integer division of dataframe and other, element-wise (binary operator <i>rfloordiv</i> ).
<code>DataFrame.rmod</code> (other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator <i>rmod</i> ).
<code>DataFrame.rmul</code> (other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator <i>rmul</i> ).
<code>DataFrame.rpow</code> (other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator <i>rpow</i> ).
<code>DataFrame.rsub</code> (other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator <i>rsub</i> ).
<code>DataFrame.rtruediv</code> (other[, axis, level, ...])	Floating division of dataframe and other, element-wise (binary operator <i>rtruediv</i> ).
<code>DataFrame.sample</code> (frac[, replace, random_state])	Random sample of items
<code>DataFrame.set_index</code> (other[, drop, sorted, ...])	Set the DataFrame index (row labels) using an existing column
<code>DataFrame.std</code> ([axis, skipna, ddof, split_every])	Return sample standard deviation over requested axis.
<code>DataFrame.sub</code> (other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator <i>sub</i> ).
<code>DataFrame.sum</code> ([axis, skipna, split_every])	Return the sum of the values for the requested axis
<code>DataFrame.tail</code> ([n, compute])	Last n rows of the dataset
<code>DataFrame.to_bag</code> ([index])	Create Dask Bag from a Dask DataFrame
<code>DataFrame.to_csv</code> (filename, **kwargs)	Store Dask DataFrame to CSV files
<code>DataFrame.to_delayed</code> ()	Create Dask Delayed objects from a Dask Dataframe
<code>DataFrame.to_hdf</code> (path_or_buf, key[, mode, ...])	Store Dask Dataframe to Hierarchical Data Format (HDF) files
<code>DataFrame.to_records</code> ([index])	Create Dask Array from a Dask Dataframe
<code>DataFrame.truediv</code> (other[, axis, level, ...])	Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>DataFrame.values</code>	Return a dask.array of the values of this dataframe
<code>DataFrame.var</code> ([axis, skipna, ddof, split_every])	Return unbiased variance over requested axis.
<code>DataFrame.visualize</code> ([filename, format, ...])	Render the computation of this object’s task graph using graphviz.
<code>DataFrame.where</code> (cond[, other])	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is True and otherwise are from <i>other</i> .

## Series

<code>Series</code> (dsk, name, meta, divisions)	Parallel Pandas Series
<code>Series.add</code> (other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator <i>add</i> ).
<code>Series.align</code> (other[, join, axis, fill_value])	Align two objects on their axes with the
<code>Series.all</code> ([axis, skipna, split_every])	Return whether all elements are True over requested axis
<code>Series.any</code> ([axis, skipna, split_every])	Return whether any element is True over requested axis
<code>Series.append</code> (other)	Concatenate two or more Series.
<code>Series.apply</code> (func[, convert_dtype, meta, args])	Parallel version of pandas.Series.apply
<code>Series.astype</code> (dtype)	Cast a pandas object to a specified dtype <i>dtype</i> .
<code>Series.autocorr</code> ([lag, split_every])	Lag-N autocorrelation
<code>Series.between</code> (left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.

Continued on next page

Table 4.19 – continued from previous page

<code>Series.bfill([axis, limit])</code>	Synonym for <code>DataFrame.fillna(method='bfill')</code>
<code>Series.cat</code>	
<code>Series.clear_divisions()</code>	Forget division information
<code>Series.clip([lower, upper, out])</code>	Trim values at input threshold(s).
<code>Series.clip_lower(threshold)</code>	Return copy of the input with values below given value(s) truncated.
<code>Series.clip_upper(threshold)</code>	Return copy of input with values above given value(s) truncated.
<code>Series.compute(**kwargs)</code>	Compute this dask collection
<code>Series.copy()</code>	Make a copy of the dataframe
<code>Series.corr(other[, method, min_periods, ...])</code>	Compute correlation with <i>other</i> Series, excluding missing values
<code>Series.count([split_every])</code>	Return number of non-NA/null observations in the Series
<code>Series.cov(other[, min_periods, split_every])</code>	Compute covariance with Series, excluding missing values
<code>Series.cummax([axis, skipna])</code>	Return cumulative max over requested axis.
<code>Series.cummin([axis, skipna])</code>	Return cumulative minimum over requested axis.
<code>Series.cumprod([axis, skipna])</code>	Return cumulative product over requested axis.
<code>Series.cumsum([axis, skipna])</code>	Return cumulative sum over requested axis.
<code>Series.describe([split_every])</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>Series.diff([periods, axis])</code>	1st discrete difference of object
<code>Series.div(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>Series.drop_duplicates([split_every, split_out])</code>	Return DataFrame with duplicate rows removed, optionally only
<code>Series.dropna()</code>	Return Series without null values
<code>Series.dt</code>	
<code>Series.dtype</code>	Return data type
<code>Series.eq(other[, level, axis])</code>	Equal to of series and other, element-wise (binary operator <i>eq</i> ).
<code>Series.ffill([axis, limit])</code>	Synonym for <code>DataFrame.fillna(method='ffill')</code>
<code>Series.fillna([value, method, limit, axis])</code>	Fill NA/NaN values using the specified method
<code>Series.first(offset)</code>	Convenience method for subsetting initial periods of time series data based on a date offset.
<code>Series.floordiv(other[, level, fill_value, axis])</code>	Integer division of series and other, element-wise (binary operator <i>floordiv</i> ).
<code>Series.ge(other[, level, axis])</code>	Greater than or equal to of series and other, element-wise (binary operator <i>ge</i> ).
<code>Series.get_partition(n)</code>	Get a dask DataFrame/Series representing the <i>nth</i> partition.
<code>Series.groupby([by])</code>	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
<code>Series.gt(other[, level, axis])</code>	Greater than of series and other, element-wise (binary operator <i>gt</i> ).
<code>Series.head([n, npartitions, compute])</code>	First n rows of the dataset
<code>Series.idxmax([axis, skipna, split_every])</code>	Return index of first occurrence of maximum over requested axis.

Continued on next page

Table 4.19 – continued from previous page

<code>Series.idxmin([axis, skipna, split_every])</code>	Return index of first occurrence of minimum over requested axis.
<code>Series.isin(values)</code>	Return a boolean <code>Series</code> showing whether each element in the <code>Series</code> is exactly contained in the passed sequence of <code>values</code> .
<code>Series.isnull()</code>	Return a boolean same-sized object indicating if the values are NA.
<code>Series.iteritems()</code>	Lazily iterate over (index, value) tuples
<code>Series.known_divisions</code>	Whether divisions are already known
<code>Series.last(offset)</code>	Convenience method for subsetting final periods of time series data based on a date offset.
<code>Series.le(other[, level, axis])</code>	Less than or equal to of series and other, element-wise (binary operator <i>le</i> ).
<code>Series.loc</code>	Purely label-location based indexer for selection by label.
<code>Series.lt(other[, level, axis])</code>	Less than of series and other, element-wise (binary operator <i>lt</i> ).
<code>Series.map(arg[, na_action, meta])</code>	Map values of Series using input correspondence (which can be
<code>Series.map_overlap(func, before, after, ...)</code>	Apply a function to each partition, sharing rows with adjacent partitions.
<code>Series.map_partitions(func, *args, **kwargs)</code>	Apply Python function on each DataFrame partition.
<code>Series.mask(cond[, other])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is False and otherwise are from <i>other</i> .
<code>Series.max([axis, skipna, split_every])</code>	This method returns the maximum of the values in the object.
<code>Series.mean([axis, skipna, split_every])</code>	Return the mean of the values for the requested axis
<code>Series.memory_usage([index, deep])</code>	Memory usage of the Series
<code>Series.min([axis, skipna, split_every])</code>	This method returns the minimum of the values in the object.
<code>Series.mod(other[, level, fill_value, axis])</code>	Modulo of series and other, element-wise (binary operator <i>mod</i> ).
<code>Series.mul(other[, level, fill_value, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<code>Series.nbytes</code>	Number of bytes
<code>Series.ndim</code>	Return dimensionality
<code>Series.ne(other[, level, axis])</code>	Not equal to of series and other, element-wise (binary operator <i>ne</i> ).
<code>Series.nlargest([n, split_every])</code>	Return the largest <i>n</i> elements.
<code>Series.notnull()</code>	Return a boolean same-sized object indicating if the values are not NA.
<code>Series.nsmallest([n, split_every])</code>	Return the smallest <i>n</i> elements.
<code>Series.nunique([split_every])</code>	Return number of unique elements in the object.
<code>Series.nunique_approx([split_every])</code>	Approximate number of unique rows.
<code>Series.persist(**kwargs)</code>	Persist this dask collection into memory
<code>Series.pipe(func, *args, **kwargs)</code>	Apply <code>func(self, *args, **kwargs)</code>
<code>Series.pow(other[, level, fill_value, axis])</code>	Exponential power of series and other, element-wise (binary operator <i>pow</i> ).
<code>Series.prod([axis, skipna, split_every])</code>	Return the product of the values for the requested axis
<code>Series.quantile(q)</code>	Approximate quantiles of Series

Continued on next page

Table 4.19 – continued from previous page

<code>Series.radd</code> ( <i>other</i> [, <i>level</i> , <i>fill_value</i> , <i>axis</i> ])	Addition of series and other, element-wise (binary operator <i>radd</i> ).
<code>Series.random_split</code> ( <i>frac</i> [, <i>random_state</i> ])	Pseudorandomly split dataframe into different pieces row-wise
<code>Series.rdiv</code> ( <i>other</i> [, <i>level</i> , <i>fill_value</i> , <i>axis</i> ])	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<code>Series.reduction</code> ( <i>chunk</i> [, <i>aggregate</i> , ...])	Generic row-wise reductions.
<code>Series.repartition</code> ([ <i>divisions</i> , <i>npartitions</i> , ...])	Repartition dataframe along new divisions
<code>Series.rename</code> ([ <i>index</i> , <i>inplace</i> , <i>sorted_index</i> ])	Alter Series index labels or name
<code>Series.resample</code> ( <i>rule</i> [, <i>how</i> , <i>closed</i> , <i>label</i> ])	Convenience method for frequency conversion and resampling of time series.
<code>Series.reset_index</code> ([ <i>drop</i> ])	Reset the index to the default index.
<code>Series.rolling</code> ( <i>window</i> [, <i>min_periods</i> , <i>freq</i> , ...])	Provides rolling transformations.
<code>Series.round</code> ([ <i>decimals</i> ])	Round each value in a Series to the given number of decimals.
<code>Series.sample</code> ( <i>frac</i> [, <i>replace</i> , <i>random_state</i> ])	Random sample of items
<code>Series.sem</code> ([ <i>axis</i> , <i>skipna</i> , <i>ddof</i> , <i>split_every</i> ])	Return unbiased standard error of the mean over requested axis.
<code>Series.shift</code> ([ <i>periods</i> , <i>freq</i> , <i>axis</i> ])	Shift index by desired number of periods with an optional time <i>freq</i>
<code>Series.size</code>	Size of the series
<code>Series.std</code> ([ <i>axis</i> , <i>skipna</i> , <i>ddof</i> , <i>split_every</i> ])	Return sample standard deviation over requested axis.
<code>Series.str</code>	
<code>Series.sub</code> ( <i>other</i> [, <i>level</i> , <i>fill_value</i> , <i>axis</i> ])	Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>Series.sum</code> ([ <i>axis</i> , <i>skipna</i> , <i>split_every</i> ])	Return the sum of the values for the requested axis
<code>Series.to_bag</code> ([ <i>index</i> ])	Create a Dask Bag from a Series
<code>Series.to_csv</code> ( <i>filename</i> , <b>**kwargs</b> )	Store Dask DataFrame to CSV files
<code>Series.to_delayed</code> ()	Create Dask Delayed objects from a Dask Dataframe
<code>Series.to_frame</code> ([ <i>name</i> ])	Convert Series to DataFrame
<code>Series.to_hdf</code> ( <i>path_or_buf</i> , <i>key</i> [, <i>mode</i> , ...])	Store Dask Dataframe to Hierarchical Data Format (HDF) files
<code>Series.to_parquet</code> ( <i>path</i> , <b>*args</b> , <b>**kwargs</b> )	Store Dask.dataframe to Parquet files
<code>Series.to_string</code> ([ <i>max_rows</i> ])	Render a string representation of the Series
<code>Series.to_timestamp</code> ([ <i>freq</i> , <i>how</i> , <i>axis</i> ])	Cast to DatetimeIndex of timestamps, at <i>beginning</i> of period
<code>Series.truediv</code> ( <i>other</i> [, <i>level</i> , <i>fill_value</i> , <i>axis</i> ])	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>Series.unique</code> ([ <i>split_every</i> , <i>split_out</i> ])	Return Series of unique values in the object.
<code>Series.value_counts</code> ([ <i>split_every</i> , <i>split_out</i> ])	Returns object containing counts of unique values.
<code>Series.values</code>	Return a dask.array of the values of this dataframe
<code>Series.var</code> ([ <i>axis</i> , <i>skipna</i> , <i>ddof</i> , <i>split_every</i> ])	Return unbiased variance over requested axis.
<code>Series.visualize</code> ([ <i>filename</i> , <i>format</i> , ...])	Render the computation of this object's task graph using <i>graphviz</i> .
<code>Series.where</code> ( <i>cond</i> [, <i>other</i> ])	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is True and otherwise are from <i>other</i> .

## Groupby Operations

<code>DataFrameGroupBy.aggregate(arg[, ...])</code>	Aggregate using callable, string, dict, or list of string/callables
<code>DataFrameGroupBy.apply(func[, meta])</code>	Parallel version of pandas GroupBy.apply
<code>DataFrameGroupBy.count([split_every, split_out])</code>	Compute count of group, excluding missing values
<code>DataFrameGroupBy.cumcount([axis])</code>	Number each item in each group from 0 to the length of that group - 1.
<code>DataFrameGroupBy.cumprod([axis])</code>	Cumulative product for each group
<code>DataFrameGroupBy.cumsum([axis])</code>	Cumulative sum for each group
<code>DataFrameGroupBy.get_group(key)</code>	Constructs NDFrame from group with provided name
<code>DataFrameGroupBy.max([split_every, split_out])</code>	Compute max of group values
<code>DataFrameGroupBy.mean([split_every, split_out])</code>	Compute mean of groups, excluding missing values
<code>DataFrameGroupBy.min([split_every, split_out])</code>	Compute min of group values
<code>DataFrameGroupBy.size([split_every, split_out])</code>	Compute group sizes
<code>DataFrameGroupBy.std([ddof, split_every, ...])</code>	Compute standard deviation of groups, excluding missing values
<code>DataFrameGroupBy.sum([split_every, split_out])</code>	Compute sum of group values
<code>DataFrameGroupBy.var([ddof, split_every, ...])</code>	Compute variance of groups, excluding missing values
<hr/>	
<code>SeriesGroupBy.aggregate(arg[, split_every, ...])</code>	Aggregate using callable, string, dict, or list of string/callables
<code>SeriesGroupBy.apply(func[, meta])</code>	Parallel version of pandas GroupBy.apply
<code>SeriesGroupBy.count([split_every, split_out])</code>	Compute count of group, excluding missing values
<code>SeriesGroupBy.cumcount([axis])</code>	Number each item in each group from 0 to the length of that group - 1.
<code>SeriesGroupBy.cumprod([axis])</code>	Cumulative product for each group
<code>SeriesGroupBy.cumsum([axis])</code>	Cumulative sum for each group
<code>SeriesGroupBy.get_group(key)</code>	Constructs NDFrame from group with provided name
<code>SeriesGroupBy.max([split_every, split_out])</code>	Compute max of group values
<code>SeriesGroupBy.mean([split_every, split_out])</code>	Compute mean of groups, excluding missing values
<code>SeriesGroupBy.min([split_every, split_out])</code>	Compute min of group values
<code>SeriesGroupBy.nunique([split_every, split_out])</code>	Compute group sizes
<code>SeriesGroupBy.size([split_every, split_out])</code>	Compute standard deviation of groups, excluding missing values
<code>SeriesGroupBy.sum([split_every, split_out])</code>	Compute sum of group values
<code>SeriesGroupBy.var([ddof, split_every, split_out])</code>	Compute variance of groups, excluding missing values

## Rolling Operations

<code>rolling.map_overlap(func, df, before, after, ...)</code>	Apply a function to each partition, sharing rows with adjacent partitions.
<code>Series.rolling(window[, min_periods, freq, ...])</code>	Provides rolling transformations.
<code>DataFrame.rolling(window[, min_periods, ...])</code>	Provides rolling transformations.
<hr/>	
<code>Rolling.apply(func[, args, kwargs])</code>	rolling function apply
<code>Rolling.count()</code>	rolling count of number of non-NaN
<code>Rolling.kurt()</code>	Unbiased rolling kurtosis
<code>Rolling.max()</code>	rolling maximum

Continued on next page

Table 4.23 – continued from previous page

<code>Rolling.mean()</code>	rolling mean
<code>Rolling.median()</code>	rolling median
<code>Rolling.min()</code>	rolling minimum
<code>Rolling.quantile(quantile)</code>	rolling quantile
<code>Rolling.skew()</code>	Unbiased rolling skewness
<code>Rolling.std([ddof])</code>	rolling standard deviation
<code>Rolling.sum()</code>	rolling sum
<code>Rolling.var([ddof])</code>	rolling variance

## Create DataFrames

<code>read_csv(urlpath[, blocksize, collection, ...])</code>	Read CSV files into a Dask.DataFrame
<code>read_table(urlpath[, blocksize, collection, ...])</code>	Read delimited files into a Dask.DataFrame
<code>read_parquet(path[, columns, filters, ...])</code>	Read ParquetFile into a Dask DataFrame
<code>read_hdf(pattern, key[, start, stop, ...])</code>	Read HDF files into a Dask DataFrame
<code>read_sql_table(table, uri, index_col[, ...])</code>	Create dataframe from an SQL table.
<code>from_array(x[, chunksize, columns])</code>	Read any slicable array into a Dask Dataframe
<code>from_bcolz(x[, chunksize, categorize, ...])</code>	Read BColz CTable into a Dask Dataframe
<code>from_dask_array(x[, columns])</code>	Create a Dask DataFrame from a Dask Array.
<code>from_delayed(dfs[, meta, divisions, prefix])</code>	Create Dask DataFrame from many Dask Delayed objects
<code>from_pandas(data[, npartitions, chunksize, ...])</code>	Construct a Dask DataFrame from a Pandas DataFrame
<code>dask.bag.core.Bag.to_dataframe([meta, columns])</code>	Create Dask Dataframe from a Dask Bag.

## Store DataFrames

<code>to_csv(df, filename[, name_function, ...])</code>	Store Dask DataFrame to CSV files
<code>to_parquet(df, path[, engine, compression, ...])</code>	Store Dask.dataframe to Parquet files
<code>to_hdf(df, path, key[, mode, append, get, ...])</code>	Store Dask Dataframe to Hierarchical Data Format (HDF) files
<code>to_records(df)</code>	Create Dask Array from a Dask Dataframe
<code>to_bag(df[, index])</code>	Create Dask Bag from a Dask DataFrame
<code>to_delayed(df)</code>	Create Dask Delayed objects from a Dask Dataframe

## DataFrame Methods

**class** `dask.dataframe.DataFrame` (*dsk, name, meta, divisions*)

Parallel Pandas DataFrame

Do not use this class directly. Instead use functions like `dd.read_csv`, `dd.read_parquet`, or `dd.from_pandas`.

### 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

### meta: pandas.DataFrame

An empty `pandas.DataFrame` with names, dtypes, and index matching the expected output.

**divisions: tuple of index values**

Values along which we partition our blocks on the index

**abs()**

Return an object with absolute value taken—only applicable to objects that are all numeric.

**Returns** `abs`: type of caller

**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

**align** (*other*, *join='outer'*, *axis=None*, *fill\_value=None*)

Align two objects on their axes with the specified join method for each axis Index

**Parameters** *other* : DataFrame or Series

**join** : {'outer', 'inner', 'left', 'right'}, default 'outer'

**axis** : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

**level** : int or level name, default None

Broadcast across a level, matching Index values on the passed MultiIndex level

**copy** : boolean, default True

Always returns new objects. If `copy=False` and no reindexing is required then original objects are returned.

**fill\_value** : scalar, default `np.NaN`

Value to use for missing values. Defaults to NaN, but can be any “compatible” value

**method** : str, default None

**limit** : int, default None

**fill\_axis** : {0 or ‘index’, 1 or ‘columns’}, default 0

Filling axis, method and limit

**broadcast\_axis** : {0 or ‘index’, 1 or ‘columns’}, default None

Broadcast values along this axis, if aligning two objects of different dimensions

New in version 0.17.0.

**Returns (left, right)** : (DataFrame, type of other)

Aligned objects

## Notes

Dask doesn’t support the following argument(s).

- level
- copy
- method
- limit
- fill\_axis
- broadcast\_axis

**all** (*axis=None, skipna=True, split\_every=False*)

Return whether all elements are True 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

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns all** : Series or DataFrame (if level specified)

## Notes

Dask doesn’t support the following argument(s).

- bool\_only
- level

**any** (*axis=None, skipna=True, split\_every=False*)

Return whether any element is True 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

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns** **any** : Series or DataFrame (if level specified)

### Notes

Dask doesn't support the following argument(s).

- `bool_only`
- `level`

**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

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

## 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
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

**apply** (*func*, *axis=0*, *args=()*, *meta='\_\_no\_default\_\_'*, *\*\*kws*)

Parallel version of `pandas.DataFrame.apply`

This mimics the pandas version except for the following:

1. Only `axis=1` is supported (and must be specified explicitly).
2. The user should provide output metadata via the `meta` keyword.

**Parameters func** : function

Function to apply to each column/row

**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

**meta** : pd.DataFrame, pd.Series, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**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

**See also:**

`dask.DataFrame.map_partitions`

**Examples**

```
>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': [1, 2, 3, 4, 5],
...                   'y': [1., 2., 3., 4., 5.]})
>>> ddf = dd.from_pandas(df, npartitions=2)
```

Apply a function to row-wise passing in extra arguments in `args` and `kwargs`:

```
>>> def myadd(row, a, b=1):
...     return row.sum() + a + b
>>> res = ddf.apply(myadd, axis=1, args=(2,), b=1.5)
```

By default, dask tries to infer the output metadata by running your provided function on some fake data. This works well in many cases, but can sometimes be expensive, or even fail. To avoid this, you can manually specify the output metadata with the `meta` keyword. This can be specified in many forms, for more information see `dask.dataframe.utils.make_meta`.

Here we specify the output is a Series with name 'x', and dtype `float64`:

```
>>> res = ddf.apply(myadd, axis=1, args=(2,), b=1.5, meta=('x', 'f8'))
```

In the case where the metadata doesn't change, you can also pass in the object itself directly:

```
>>> res = ddf.apply(lambda row: row + 1, axis=1, meta=ddf)
```

**applymap** (*func*, *meta*='\_\_no\_default\_\_')

Apply a function to a DataFrame that is intended to operate elementwise, i.e. like doing `map(func, series)` for each series in the DataFrame

**Parameters** *func* : function

Python function, returns a single value from a single value

**Returns** *applied* : DataFrame

**See also:**

[`DataFrame.apply`](#) For operations on rows/columns

## Examples

```
>>> df = pd.DataFrame(np.random.randn(3, 3))
>>> df
   0         1         2
0 -0.029638  1.081563  1.280300
1  0.647747  0.831136 -1.549481
2  0.513416 -0.884417  0.195343
>>> df = df.applymap(lambda x: '%.2f' % x)
>>> df
   0         1         2
0 -0.03         1.08         1.28
1  0.65         0.83        -1.55
2  0.51        -0.88         0.20
```

**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.

**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

For python 3.6 and above, the columns are inserted in the order of `**kwargs`. For python 3.5 and earlier, since `**kwargs` is unordered, 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 a pandas object to a specified dtype *dtype*.

**Parameters** **dtype** : data type, or dict of column name -> data type

Use a `numpy.dtype` or Python type to cast entire pandas object to the same type. Alternatively, use `{col: dtype, ...}`, where `col` is a column label and `dtype` is a `numpy.dtype` or Python type to cast one or more of the DataFrame's columns to column-specific types.

**copy** : bool, default True.

Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

**errors** : {'raise', 'ignore'}, default 'raise'.

Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

**raise\_on\_error** : raise on invalid input

Deprecated since version 0.20.0: Use `errors` instead

**kwargs** : keyword arguments to pass on to the constructor

**Returns** **casted** : type of caller

**See also:**

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Convert argument to a numeric type.

`numpy.ndarray.astype` Cast a numpy array to a specified type.

**Notes**

Dask doesn't support the following argument(s).

- `copy`
- `errors`

**Examples**

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int', copy=False)
>>> s2[0] = 10
>>> s1 # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

**bfill** (*axis=None, limit=None*)

Synonym for `DataFrame.fillna(method='bfill')`

### Notes

Dask doesn't support the following argument(s).

- inplace
- downcast

**categorize** (*df, columns=None, index=None, split\_every=None, \*\*kwargs*)

Convert columns of the DataFrame to category dtype.

**Parameters** **columns** : list, optional

A list of column names to convert to categoricals. By default any column with an object dtype is converted to a categorical, and any unknown categoricals are made known.

**index** : bool, optional

Whether to categorize the index. By default, object indices are converted to categorical, and unknown categorical indices are made known. Set True to always categorize the index, False to never.

**split\_every** : int, optional

Group partitions into groups of this size while performing a tree-reduction. If set to False, no tree-reduction will be used. Default is 16.

**kwargs**

Keyword arguments are passed on to compute.

**clear\_divisions** ()

Forget division information

**clip** (*lower=None, upper=None, out=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.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns** **clipped** : Series

### Notes

Dask doesn't support the following argument(s).

- axis
- inplace

## 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
```

### `clip_lower` (*threshold*)

Return copy of the input with values below given value(s) truncated.

**Parameters** `threshold` : float or array\_like

**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns** `clipped` : same type as input

**See also:**

`clip`

### Notes

Dask doesn't support the following argument(s).

- axis
- inplace

**clip\_upper** (*threshold*)

Return copy of input with values above given value(s) truncated.

**Parameters** **threshold** : float or array\_like

**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns** **clipped** : same type as input

**See also:**

*clip*

## Notes

Dask doesn't support the following argument(s).

- axis
- inplace

**combine** (*other, func, fill\_value=None, overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

**Parameters** **other** : DataFrame

**func** : function

Function that takes two series as inputs and return a Series or a scalar

**fill\_value** : scalar value

**overwrite** : boolean, default True

If True then overwrite values for common keys in the calling frame

**Returns** **result** : DataFrame

**See also:**

*DataFrame.combine\_first* Combine two DataFrame objects and default to non-null values in frame calling the method

## Examples

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
  A  B
```

```
0 0 3
1 0 3
```

**combine\_first** (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

**Parameters other** : DataFrame

**Returns combined** : DataFrame

**See also:**

[\*DataFrame.combine\*](#) Perform series-wise operation on two DataFrames using a given function

**Examples**

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0  1
0  1  4.0
```

**compute** (*\*\*kwargs*)

Compute this dask collection

This turns a lazy Dask collection into its in-memory equivalent. For example a Dask.array turns into a NumPy array and a Dask.dataframe turns into a Pandas dataframe. The entire dataset must fit into memory before calling this operation.

**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.

**See also:**

`dask.base.compute`

**copy** ()

Make a copy of the dataframe

This is strictly a shallow copy of the underlying computational graph. It does not affect the underlying data

**corr** (*method='pearson', min\_periods=None, split\_every=False*)

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, split\_every=False*)

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 support the following argument(s).

- `level`
- `numeric_only`

**cov** (*min\_periods=None, split\_every=False*)

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 max 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

**See also:**

`pandas.core.window.Expanding.max` Similar functionality but ignores NaN values.

**cummin** (*axis=None, skipna=True*)

Return cumulative minimum 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

**See also:**

`pandas.core.window.Expanding.min` Similar functionality but ignores NaN values.

**cumprod** (*axis=None, skipna=True*)

Return cumulative product 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

**See also:**

`pandas.core.window.Expanding.prod` Similar functionality but ignores NaN values.

**cumsum** (*axis=None, skipna=True*)

Return cumulative sum 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

**See also:**

`pandas.core.window.Expanding.sum` Similar functionality but ignores NaN values.

**describe** (*split\_every=False*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters** **percentiles** : list-like of numbers, optional

The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

**include** : 'all', list-like of dtypes or None (default), optional

A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- `'all'` : All columns of the input will be included in the output.
- A list-like of `dtypes` : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- `None` (default) : The result will include all numeric columns.

**exclude** : list-like of `dtypes` or `None` (default), optional,

A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of `dtypes` : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- `None` (default) : The result will exclude nothing.

**Returns** summary: `Series/DataFrame` of summary statistics

**See also:**

*`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`*

## Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as `lower`, `50` and `upper` percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

## Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count      4
unique     3
top        a
freq       2
dtype: object
```

Describing a timestamp Series.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                3
unique                2
top      2010-01-01 00:00:00
freq                2
first  2000-01-01 00:00:00
last   2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                    'numeric': [1, 2, 3],
...                    'categorical': pd.Categorical(['d', 'e', 'f'])
...                    })
>>> df.describe()
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a DataFrame regardless of data type.

```
>>> df.describe(include='all')
      categorical  numeric  object
```

count	3	3.0	3
unique	3	NaN	3
top	f	NaN	c
freq	1	NaN	1
mean	NaN	2.0	NaN
std	NaN	1.0	NaN
min	NaN	1.0	NaN
25%	NaN	1.5	NaN
50%	NaN	2.0	NaN
75%	NaN	2.5	NaN
max	NaN	3.0	NaN

Describing a column from a DataFrame by accessing it as an attribute.

```
>>> df.numeric.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%       1.5
50%       2.0
75%       2.5
max        3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a DataFrame description.

```
>>> df.describe(include=[np.number])
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%       1.5
50%       2.0
75%       2.5
max        3.0
```

Including only string columns in a DataFrame description.

```
>>> df.describe(include=[np.object])
      object
count      3
unique     3
top        c
freq       1
```

Including only categorical columns from a DataFrame description.

```
>>> df.describe(include=['category'])
      categorical
count          3
unique         3
top            f
freq           1
```

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
           categorical  object
count              3      3
unique             3      3
top               f      c
freq              1      1
```

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
           categorical  numeric
count              3      3.0
unique             3      NaN
top               f      NaN
freq              1      NaN
mean             NaN      2.0
std              NaN      1.0
min              NaN      1.0
25%              NaN      1.5
50%              NaN      2.0
75%              NaN      2.5
max              NaN      3.0
```

**diff** (*periods=1, axis=0*)

1st discrete difference of object

**Parameters** **periods** : int, default 1

Periods to shift for forming difference

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Take difference over rows (0) or columns (1).

**Returns** **differed** : DataFrame

**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*, *errors='raise'*)

Return new object with labels in requested axis removed.

**Parameters labels** : single label or list-like

Index or column labels to drop.

**axis** : int or axis name

Whether to drop labels from the index (0 / 'index') or columns (1 / 'columns').

**index, columns** : single label or list-like

Alternative to specifying *axis* (*labels*, *axis=1* is equivalent to *columns=labels*).

New in version 0.21.0.

**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.

**Returns dropped** : type of caller

## Notes

Specifying both *labels* and *index* or *columns* will raise a ValueError.

## Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
                      columns=['A', 'B', 'C', 'D'])
>>> df
   A  B  C  D
0  0  1  2  3
1  4  5  6  7
2  8  9 10 11
```

### Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A  D
0  0  3
1  4  7
2  8 11
```

```
>>> df.drop(columns=['B', 'C'])
   A  D
0  0  3
1  4  7
2  8 11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B  C  D
2  8  9 10 11
```

**drop\_duplicates** (*split\_every=None, split\_out=1, \*\*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.

**inplace** : boolean, default False

Whether to drop duplicates in place or to return a copy

**Returns** **deduplicated** : DataFrame

## Notes

Dask doesn't support the following argument(s).

- subset
- keep
- inplace

**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 support the following argument(s).

- axis
- thresh
- inplace

## Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0], [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5]],
...                    columns=list('ABCD'))
>>> df
   A    B    C    D
0 NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
```

Drop the columns where all elements are nan:

```
>>> df.dropna(axis=1, how='all')
   A    B    D
0 NaN  2.0  0
1  3.0  4.0  1
2  NaN  NaN  5
```

Drop the columns where any of the elements is nan

```
>>> df.dropna(axis=1, how='any')
   D
0  0
1  1
2  5
```

Drop the rows where all of the elements are nan (there is no row to drop, so df stays the same):

```
>>> df.dropna(axis=0, how='all')
   A    B    C    D
0 NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
```

Keep only the rows with at least 2 non-na values:

```
>>> df.dropna(thresh=2)
   A    B    C    D
0 NaN  2.0 NaN    0
1  3.0  4.0 NaN    1
```

**dtypes**

Return data types

**eq** (*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods `eq`

**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, default False

If the expression contains an assignment, whether to perform the operation in-place and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

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')
```

**ffill** (*axis=None*, *limit=None*)

Synonym for `DataFrame.fillna(method='ffill')`

**Notes**

Dask doesn't support the following argument(s).

- inplace
- downcast

**fillna** (*value=None, method=None, limit=None, axis=None*)

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 or 'index', 1 or 'columns'}

**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. Must be greater than 0 if not None.

**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** : DataFrame

**See also:**

`reindex`, `asfreq`

## Notes

Dask doesn't support the following argument(s).

- inplace
- downcast

## Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                   [3, 4, np.nan, 1],
...                   [np.nan, np.nan, np.nan, 5],
...                   [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
```

```
>>> df
   A    B    C    D
0 NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
   A    B    C    D
0  0.0  2.0  0.0  0
1  3.0  4.0  0.0  1
2  0.0  0.0  0.0  5
3  0.0  3.0  0.0  4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
   A    B    C    D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  3.0  4.0 NaN  5
3  3.0  3.0 NaN  4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  2.0  1
2  0.0  1.0  2.0  5
3  0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0 NaN  1
2  NaN  1.0 NaN  5
3  NaN  3.0 NaN  4
```

#### **first** (*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

**Parameters** `offset`: string, DateOffset, dateutil.relativedelta

**Returns** `subset`: type of caller

#### **Examples**

`ts.first('10D')` -> First 10 days

**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

**ge** (*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods `ge`

**get\_dtype\_counts** ()

Return the counts of dtypes in this object.

**get\_ftype\_counts** ()

Return the counts of ftypes in this object.

**get\_partition** (*n*)

Get a dask DataFrame/Series representing the *n*th partition.

**groupby** (*by=None*, *\*\*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, str, or iterable

Used to determine the groups for the `groupby`. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A str or list of str's may be passed to group by the columns in `self`

**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 support the following argument(s).

- `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()
```

**gt** (*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods `gt`

**head** (*n=5*, *npartitions=1*, *compute=True*)

First `n` rows of the dataset

**Parameters** `n` : int, optional

The number of rows to return. Default is 5.

**npartitions** : int, optional

Elements are only taken from the first `npartitions`, with a default of 1. If there are fewer than `n` rows in the first `npartitions` a warning will be raised and any found rows returned. Pass -1 to use all partitions.

**compute** : bool, optional

Whether to compute the result, default is True.

**idxmax** (*axis=None, skipna=True, split\_every=False*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

**Parameters axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns idxmax** : Series

**Raises ValueError**

- If the row/column is empty

**See also:**

`Series.idxmax`

### Notes

This method is the DataFrame version of `ndarray.argmax`.

**idxmin** (*axis=None, skipna=True, split\_every=False*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

**Parameters axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns idxmin** : Series

**Raises ValueError**

- If the row/column is empty

**See also:**

`Series.idxmin`

### Notes

This method is the DataFrame version of `ndarray.argmin`.

**index**

Return dask Index instance

**info** (*buf=None, verbose=False, memory\_usage=False*)

Concise summary of a Dask DataFrame.

**isin** (*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

**Parameters values** : iterable, Series, DataFrame or dictionary

The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

**Returns** DataFrame of booleans

## Examples

When *values* is a list:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
   A      B
0  True  True
1 False False
2  True  False
```

When *values* is a dict:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
   A      B
0  True  False # Note that B didn't match the 1 here.
1 False  True
2  True  True
```

When *values* is a Series or DataFrame:

```
>>> df = DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
   A      B
0  True  False
1 False  False # Column A in `other` has a 3, but not at index 1.
2  True  True
```

### `isnull()`

Return a boolean same-sized object indicating if the values are NA.

**See also:**

**DataFrame.notna** boolean inverse of `isna`

**DataFrame.isnull** alias of `isna`

**isna** top-level `isna`

### `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

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

## 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*, *shuffle=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) in the caller to join on the index in other, otherwise joins index-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' }, default: 'left'

How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

**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, the order of the join key depends on the join type (how keyword)

**Returns** **joined** : DataFrame

**See also:**

[\*DataFrame.merge\*](#) For column(s)-on-columns(s) operations

## Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

## Examples

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
   A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
   B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>
   A key_caller  B key_other
0  A0          K0  B0          K0
1  A1          K1  B1          K1
2  A2          K2  B2          K2
3  A3          K3  NaN         NaN
4  A4          K4  NaN         NaN
5  A5          K5  NaN         NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>
   key  A  B
K0  A0  B0
K1  A1  B1
K2  A2  B2
K3  A3  NaN
K4  A4  NaN
K5  A5  NaN
```

Another option to join using the key columns is to use the on parameter. DataFrame.join always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```

>>>      A key   B
0  A0  K0   B0
1  A1  K1   B1
2  A2  K2   B2
3  A3  K3  NaN
4  A4  K4  NaN
5  A5  K5  NaN

```

**known\_divisions**

Whether divisions are already known

**last** (*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

**Parameters** *offset* : string, DateOffset, dateutil.relativedelta

**Returns** *subset* : type of caller

**Examples**

`ts.last('5M')` -> Last 5 months

**le** (*other, axis='columns', level=None*)

Wrapper for flexible comparison methods `le`

**loc**

Purely label-location based indexer for selection by label.

```

>>> df.loc["b"]
>>> df.loc["b":"d"]

```

**lt** (*other, axis='columns', level=None*)

Wrapper for flexible comparison methods `lt`

**map\_overlap** (*func, before, after, \*args, \*\*kwargs*)

Apply a function to each partition, sharing rows with adjacent partitions.

This can be useful for implementing windowing functions such as `df.rolling(...).mean()` or `df.diff()`.

**Parameters** *func* : function

Function applied to each partition.

**before** : int

The number of rows to prepend to partition *i* from the end of partition *i - 1*.

**after** : int

The number of rows to append to partition *i* from the beginning of partition *i + 1*.

**args, kwargs** :

Arguments and keywords to pass to the function. The partition will be the first argument, and these will be passed *after*.

**meta** : pd.DataFrame, pd.Series, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

## Notes

Given positive integers `before` and `after`, and a function `func`, `map_overlap` does the following:

1. Prepend `before` rows to each partition `i` from the end of partition `i - 1`. The first partition has no rows prepended.
2. Append `after` rows to each partition `i` from the beginning of partition `i + 1`. The last partition has no rows appended.
3. Apply `func` to each partition, passing in any extra `args` and `kwargs` if provided.
4. Trim `before` rows from the beginning of all but the first partition.
5. Trim `after` rows from the end of all but the last partition.

Note that the index and divisions are assumed to remain unchanged.

## Examples

Given a `DataFrame`, `Series`, or `Index`, such as:

```
>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': [1, 2, 4, 7, 11],
...                   'y': [1., 2., 3., 4., 5.]})
>>> ddf = dd.from_pandas(df, npartitions=2)
```

A rolling sum with a trailing moving window of size 2 can be computed by overlapping 2 rows before each partition, and then mapping calls to `df.rolling(2).sum()`:

```
>>> ddf.compute()
   x    y
0  1  1.0
1  2  2.0
2  4  3.0
3  7  4.0
4 11  5.0
>>> ddf.map_overlap(lambda df: df.rolling(2).sum(), 2, 0).compute()
   x    y
0  NaN NaN
1  3.0 3.0
2  6.0 5.0
3 11.0 7.0
4 18.0 9.0
```

The pandas `diff` method computes a discrete difference shifted by a number of periods (can be positive or negative). This can be implemented by mapping calls to `df.diff` to each partition after prepending/appending that many rows, depending on sign:

```

>>> def diff(df, periods=1):
...     before, after = (periods, 0) if periods > 0 else (0, -periods)
...     return df.map_overlap(lambda df: df.diff(periods),
...                           periods, 0, periods=periods)
>>> diff(ddf, 1).compute()
   x    y
0 NaN NaN
1 1.0 1.0
2 2.0 1.0
3 3.0 1.0
4 4.0 1.0

```

If you have a `DatetimeIndex`, you can use a `timedelta` for time- based windows. `>>> ts = pd.Series(range(10), index=pd.date_range('2017', periods=10)) >>> dts = dd.from_pandas(ts, npartitions=2) >>> dts.map_overlap(lambda df: df.rolling('2D').sum(), ... pd.Timedelta('2D'), 0).compute()`  
2017-01-01 0.0 2017-01-02 1.0 2017-01-03 3.0 2017-01-04 5.0 2017-01-05 7.0 2017-01-06 9.0 2017-01-07 11.0 2017-01-08 13.0 2017-01-09 15.0 2017-01-10 17.0 dtype: float64

**map\_partitions** (*func*, \**args*, \*\**kwargs*)

Apply Python function on each DataFrame partition.

Note that the index and divisions are assumed to remain unchanged.

**Parameters** *func* : function

Function applied to each partition.

**args, kwargs** :

Arguments and keywords to pass to the function. The partition will be the first argument, and these will be passed *after*.

**meta** : `pd.DataFrame`, `pd.Series`, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of {name: dtype} or iterable of (name, dtype) can be provided. Instead of a series, a tuple of (name, dtype) can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing *meta* is recommended. For more information, see `dask.dataframe.utils.make_meta`.

## Examples

Given a `DataFrame`, `Series`, or `Index`, such as:

```

>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': [1, 2, 3, 4, 5],
...                   'y': [1., 2., 3., 4., 5.]})
>>> ddf = dd.from_pandas(df, npartitions=2)

```

One can use `map_partitions` to apply a function on each partition. Extra arguments and keywords can optionally be provided, and will be passed to the function after the partition.

Here we apply a function with arguments and keywords to a `DataFrame`, resulting in a `Series`:

```
>>> def myadd(df, a, b=1):
...     return df.x + df.y + a + b
>>> res = ddf.map_partitions(myadd, 1, b=2)
>>> res.dtype
dtype('float64')
```

By default, dask tries to infer the output metadata by running your provided function on some fake data. This works well in many cases, but can sometimes be expensive, or even fail. To avoid this, you can manually specify the output metadata with the `meta` keyword. This can be specified in many forms, for more information see `dask.dataframe.utils.make_meta`.

Here we specify the output is a Series with no name, and dtype `float64`:

```
>>> res = ddf.map_partitions(myadd, 1, b=2, meta=(None, 'f8'))
```

Here we map a function that takes in a DataFrame, and returns a DataFrame with a new column:

```
>>> res = ddf.map_partitions(lambda df: df.assign(z=df.x * df.y))
>>> res.dtypes
x      int64
y      float64
z      float64
dtype: object
```

As before, the output metadata can also be specified manually. This time we pass in a dict, as the output is a DataFrame:

```
>>> res = ddf.map_partitions(lambda df: df.assign(z=df.x * df.y),
...                          meta={'x': 'i8', 'y': 'f8', 'z': 'f8'})
```

In the case where the metadata doesn't change, you can also pass in the object itself directly:

```
>>> res = ddf.map_partitions(lambda df: df.head(), meta=df)
```

Also note that the index and divisions are assumed to remain unchanged. If the function you're mapping changes the index/divisions, you'll need to clear them afterwards:

```
>>> ddf.map_partitions(func).clear_divisions()
```

### **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-like, or callable

Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. 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

Entries where *cond* is True are replaced with corresponding value from *other*. 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

**errors** : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**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)

Deprecated since version 0.21.0.

**Returns** **wh** : same type as caller

**See also:**

`DataFrame.where()`

## Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in [indexing](#).

## Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
```

```
3 NaN
4 NaN
```

```
>>> s.where(s > 1, 10)
0 10.0
1 10.0
2  2.0
3  3.0
4  4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

**max** (*axis=None, skipna=True, split\_every=False*)

**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 when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** *max* : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- level
- numeric\_only

**mean** (*axis=None, skipna=True, split\_every=False*)

Return the mean of the values for the requested axis

**Parameters axis** : {index (0), columns (1)}

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns mean** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- level
- numeric\_only

**memory\_usage** (*index=True, deep=False*)

Memory usage of DataFrame columns.

**Parameters index** : bool

Specifies whether to include memory usage of DataFrame's index in returned Series. If *index=True* (default is False) the first index of the Series is *Index*.

**deep** : bool

Introspect the data deeply, interrogate *object* dtypes for system-level memory consumption

**Returns sizes** : Series

A series with column names as index and memory usage of columns with units of bytes.

**See also:**

`numpy.ndarray.nbytes`

## Notes

Memory usage does not include memory consumed by elements that are not components of the array if `deep=False`

**merge** (*right*, *how*='inner', *on*=None, *left\_on*=None, *right\_on*=None, *left\_index*=False, *right\_index*=False, *suffixes*=('\_x', '\_y'), *indicator*=False, *npartitions*=None, *shuffle*=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, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

**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. If False, the order of the join keys depends on the join type (how keyword)

**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.

**validate** : string, default None

If specified, checks if merge is of specified type.

- “one\_to\_one” or “1:1”: check if merge keys are unique in both left and right datasets.
- “one\_to\_many” or “1:m”: check if merge keys are unique in left dataset.
- “many\_to\_one” or “m:1”: check if merge keys are unique in right dataset.
- “many\_to\_many” or “m:m”: allowed, but does not result in checks.

New in version 0.21.0.

**Returns merged** : DataFrame

The output type will be the same as ‘left’, if it is a subclass of DataFrame.

**See also:**

`merge_ordered`, `merge_asof`

## Notes

Dask doesn’t support the following argument(s).

- sort
- copy
- validate

## 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, split\_every=False*)

**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 when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** **min** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the 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

**ne** (*other*, *axis*='columns', *level*=None)

Wrapper for flexible comparison methods *ne*

**nlargest** (*n*=5, *columns*=None, *split\_every*=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'}, default 'first'

Where there are duplicate values: - *first* : take the first occurrence. - *last* : take the last occurrence.

**Returns** DataFrame

## Notes

Dask doesn't support the 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 NA.

#### See also:

`DataFrame.isna` boolean inverse of `notna`

`DataFrame.notnull` alias of `notna`

`notna` top-level `notna`

### `npartitions`

Return number of partitions

### `nsmallest` (*n=5, columns=None, split\_every=None*)

Get the rows of a DataFrame sorted by the *n* smallest 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'}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

**Returns** DataFrame

## Notes

Dask doesn't support the 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.nsmallest(3, 'a')
   a  b  c
4 -1  e  4
0  1  a  1
2  8  d NaN
```

**nunique\_approx** (*split\_every=None*)

Approximate number of unique rows.

This method uses the HyperLogLog algorithm for cardinality estimation to compute the approximate number of unique rows. The approximate error is 0.406%.

**Parameters** *split\_every* : int, optional

Group partitions into groups of this size while performing a tree-reduction. If set to False, no tree-reduction will be used. Default is 8.

**Returns** a float representing the approximate number of elements

**persist** (*\*\*kwargs*)

Persist this dask collection into memory

This turns a lazy Dask collection into a Dask collection with the same metadata, but now with the results fully computed or actively computing in the background.

**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.

**Returns** New dask collections backed by in-memory data

**See also:**

`dask.base.persist`

**pipe** (*func, \*args, \*\*kwargs*)

Apply `func(self, *args, **kwargs)`

**Parameters** *func* : function

function to apply to the NDFrame. *args*, and *kwargs* are passed into *func*. Alternatively a (callable, *data\_keyword*) tuple where *data\_keyword* is a string indicating the keyword of callable that expects the NDFrame.

**args** : iterable, optional

positional arguments passed into *func*.

**kwargs** : mapping, optional

a dictionary of keyword arguments passed into `func`.

**Returns object** : the return type of `func`.

**See also:**

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

## Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe((f, 'arg2'), arg1=a, arg3=c)
... )
```

**pivot\_table** (*index=None, columns=None, values=None, aggfunc='mean'*)

Create a spreadsheet-style pivot table as a DataFrame. Target `columns` must have category dtype to infer result's columns. `index`, `columns`, `values` and `aggfunc` must be all scalar.

**Parameters values** : scalar

column to aggregate

**index** : scalar

column to be index

**columns** : scalar

column to be columns

**aggfunc** : {'mean', 'sum', 'count'}, default 'mean'

**Returns table** : DataFrame

**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

**prod** (*axis=None, skipna=True, split\_every=False*)

Return the product of the values for the requested axis

**Parameters axis** : {index (0), columns (1)}

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 1. This means the sum or product of an all-NA or empty series is NaN.

**Returns prod** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- level
- numeric\_only
- min\_count

## Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

**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

**query** (*expr, \*\*kwargs*)

Filter dataframe with complex expression

Blocked version of `pd.DataFrame.query`

This is like the sequential version except that this will also happen in many threads. This may conflict with `numexpr` which will use multiple threads itself. We recommend that you set `numexpr` to use a single thread

```
import numexpr
numexpr.set_nthreads(1)
```

**See also:**

`pandas.DataFrame.query`

**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** (*frac, random\_state=None*)

Pseudorandomly split dataframe into different pieces row-wise

**Parameters frac** : list

List of floats that should sum to one.

**random\_state**: int or `np.random.RandomState`

If int create a new RandomState with this as the seed

**Otherwise draw from the passed RandomState**

**See also:**

`dask.DataFrame.sample`

## 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

**reduction** (*chunk*, *aggregate=None*, *combine=None*, *meta='\_\_no\_default\_\_'*, *token=None*, *split\_every=None*, *chunk\_kwargs=None*, *aggregate\_kwargs=None*, *combine\_kwargs=None*, *\*\*kwargs*)

Generic row-wise reductions.

**Parameters** **chunk** : callable

Function to operate on each partition. Should return a `pandas.DataFrame`, `pandas.Series`, or a scalar.

**aggregate** : callable, optional

Function to operate on the concatenated result of `chunk`. If not specified, defaults to `chunk`. Used to do the final aggregation in a tree reduction.

The input to `aggregate` depends on the output of `chunk`. If the output of `chunk` is a:

- scalar: Input is a `Series`, with one row per partition.
- `Series`: Input is a `DataFrame`, with one row per partition. Columns are the rows in the output series.
- `DataFrame`: Input is a `DataFrame`, with one row per partition. Columns are the columns in the output dataframes.

Should return a `pandas.DataFrame`, `pandas.Series`, or a scalar.

**combine** : callable, optional

Function to operate on intermediate concatenated results of `chunk` in a tree-reduction. If not provided, defaults to `aggregate`. The input/output requirements should match that of `aggregate` described above.

**meta** : `pd.DataFrame`, `pd.Series`, `dict`, `iterable`, `tuple`, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in `dask dataframe` to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a `dict` of `{name: dtype}` or `iterable` of `(name, dtype)` can be provided. Instead of a `series`, a `tuple` of `(name, dtype)` can be used. If not provided, `dask` will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**token** : str, optional

The name to use for the output keys.

**split\_every** : int, optional

Group partitions into groups of this size while performing a tree-reduction. If set to `False`, no tree-reduction will be used, and all intermediates will be concatenated and passed to `aggregate`. Default is 8.

**chunk\_kwargs** : dict, optional

Keyword arguments to pass on to chunk only.

**aggregate\_kwargs** : dict, optional

Keyword arguments to pass on to aggregate only.

**combine\_kwargs** : dict, optional

Keyword arguments to pass on to combine only.

**kwargs** :

All remaining keywords will be passed to chunk, combine, and aggregate.

## Examples

```
>>> import pandas as pd
>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': range(50), 'y': range(50, 100)})
>>> ddf = dd.from_pandas(df, npartitions=4)
```

Count the number of rows in a DataFrame. To do this, count the number of rows in each partition, then sum the results:

```
>>> res = ddf.reduction(lambda x: x.count(),
...                    aggregate=lambda x: x.sum())
>>> res.compute()
x      50
y      50
dtype: int64
```

Count the number of rows in a Series with elements greater than or equal to a value (provided via a keyword).

```
>>> def count_greater(x, value=0):
...     return (x >= value).sum()
>>> res = ddf.x.reduction(count_greater, aggregate=lambda x: x.sum(),
...                       chunk_kwargs={'value': 25})
>>> res.compute()
25
```

Aggregate both the sum and count of a Series at the same time:

```
>>> def sum_and_count(x):
...     return pd.Series({'sum': x.sum(), 'count': x.count()})
>>> res = ddf.x.reduction(sum_and_count, aggregate=lambda x: x.sum())
>>> res.compute()
count      50
sum       1225
dtype: int64
```

Doing the same, but for a DataFrame. Here chunk returns a DataFrame, meaning the input to aggregate is a DataFrame with an index with non-unique entries for both 'x' and 'y'. We groupby the index, and sum each group to get the final result.

```
>>> def sum_and_count(x):
...     return pd.DataFrame({'sum': x.sum(), 'count': x.count()})
>>> res = ddf.reduction(sum_and_count,
...                     aggregate=lambda x: x.groupby(level=0).sum())
>>> res.compute()
   count  sum
x      50 1225
y      50 3725
```

**rename** (*index=None, columns=None*)

Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the [user guide](#) for more.

**Parameters** **mapper, index, columns** : dict-like or function, optional

dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

**axis** : int or str, optional

Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

**copy** : boolean, default True

Also copy underlying data

**inplace** : boolean, default False

Whether to return a new %(klass)s. If True then value of copy is ignored.

**level** : int or level name, default None

In case of a MultiIndex, only rename labels in the specified level.

**Returns** **renamed** : DataFrame

**See also:**

`pandas.DataFrame.rename_axis`

## Notes

Dask doesn't support the following argument(s).

- `mapper`
- `axis`
- `copy`
- `inplace`
- `level`

## Examples

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
```

	a	c
0	1	4
1	2	5
2	3	6

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
```

	a	B	c
0	1	4	
1	2	5	
2	3	6	

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
```

	a	b
0	1	4
1	2	5
2	3	6

```
>>> df.rename({1: 2, 2: 4}, axis='index')
```

	A	B
0	1	4
2	2	5
4	3	6

**repartition** (*divisions=None, npartitions=None, freq=None, force=False*)

Repartition dataframe along new divisions

**Parameters** **divisions** : list, optional

List of partitions to be used. If specified `npartitions` will be ignored.

**npartitions** : int, optional

Number of partitions of output, must be less than `npartitions` of input. Only used if `divisions` isn't specified.

**freq** : str, `pd.Timedelta`

A period on which to partition timeseries data like '7D' or '12h' or `pd.Timedelta(hours=12)`. Assumes a datetime index.

**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])
>>> df = df.repartition(freq='7d')
```

**resample** (*rule*, *how=None*, *closed=None*, *label=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the *on* or *level* keyword.

**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. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**label** : {'right', 'left'}

Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**convention** : {'start', 'end', 's', 'e'}

For PeriodIndex only, controls whether to use the start or end of *rule*

**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

**on** : string, optional

For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

**level** : string or int, optional

For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

## Notes

To learn more about the offset strings, please see [this link](#).

## 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    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00    1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
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
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))

>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using *'start'* convention. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using *'end'* convention. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
```

```

2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64

```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```

>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()

```

	a	b	c	d
time				
2000-01-01 00:00:00	0	3	6	9
2000-01-01 00:03:00	0	3	6	9
2000-01-01 00:06:00	0	3	6	9

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```

>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                      columns=['a', 'b', 'c', 'd'],
                      index=pd.MultiIndex.from_product([time, [1, 2]]))
>>> df2.resample('3T', level=0).sum()

```

	a	b	c	d
2000-01-01 00:00:00	0	6	12	18
2000-01-01 00:03:00	0	4	8	12

#### **reset\_index** (*drop=False*)

Reset the index to the default index.

Note that unlike in pandas, the reset `dask.dataframe` index will not be monotonically increasing from 0. Instead, it will restart at 0 for each partition (e.g. `index1 = [0, ..., 10]`, `index2 = [0, ...]`). This is due to the inability to statically know the full length of the index.

For DataFrame with multi-level index, returns a 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** `drop` : boolean, default False

Do not try to insert index into dataframe columns.

#### **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, str, offset

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. If using an offset or offset alias like '5D', the data must have a `DatetimeIndex`

Changed in version 0.15.0: Now accepts offsets and string offset aliases

**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.

**round** (*decimals=0*)  
Round a DataFrame to a variable number of decimal places.

New in version 0.17.0.

**Parameters decimals** : int, dict, Series

Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

**Returns** DataFrame object

**See also:**

`numpy.around`, `Series.round`

## Examples

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...                    columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
      A         B         C
first 0.028208 0.992815 0.173891
second 0.038683 0.645646 0.577595
third 0.877076 0.149370 0.491027
>>> df.round(2)
      A         B         C
first 0.03 0.99 0.17
second 0.04 0.65 0.58
third 0.88 0.15 0.49
>>> df.round({'A': 1, 'C': 2})
      A         B         C
first 0.0 0.992815 0.17
second 0.0 0.645646 0.58
third 0.9 0.149370 0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
      A  B         C
first 0.0 1 0.17
second 0.0 1 0.58
third 0.9 0 0.49
```

**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

**See also:**

`DataFrame.random_split`, `pandas.DataFrame.sample`

**select\_dtypes** (*include=None*, *exclude=None*)

Return a subset of a DataFrame including/excluding columns based on their dtype.

**Parameters** **include**, **exclude** : scalar or list-like

A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

**Returns** **subset** : DataFrame

The subset of the frame including the dtypes in *include* and excluding the dtypes in *exclude*.

**Raises** **ValueError**

- If both of *include* and *exclude* are empty
- If *include* and *exclude* have overlapping elements
- If any kind of string dtype is passed in.

## Notes

- To select all *numeric* types use the numpy dtype `numpy.number`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the [numpy dtype hierarchy](#)
- To select datetimes, use `np.datetime64`, ‘datetime’ or ‘datetime64’
- To select timedeltas, use `np.timedelta64`, ‘timedelta’ or ‘timedelta64’
- To select Pandas categorical dtypes, use ‘category’
- To select Pandas datetimetz dtypes, use ‘datetimetz’ (new in 0.20.0), or a ‘datetime64[ns, tz]’ string

## Examples

```

>>> df = pd.DataFrame({'a': np.random.randn(6).astype('f4'),
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
   a      b  c
0  0.3962  True  1
1  0.1459  False 2
2  0.2623  True  1
3  0.0764  False 2
4 -0.9703  True  1
5 -1.2094  False 2
>>> df.select_dtypes(include='bool')
   c
0  True
1  False
2  True
3  False
4  True
5  False
>>> df.select_dtypes(include=['float64'])
   c
0  1
1  2
2  1
3  2
4  1
5  2
>>> df.select_dtypes(exclude=['floating'])
   b
0  True
1  False
2  True
3  False
4  True
5  False

```

**sem** (*axis=None, skipna=None, ddof=1, split\_every=False*)

Return unbiased standard error of the mean 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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** `sem` : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- level
- numeric\_only

**set\_index** (*other, drop=True, sorted=False, npartitions=None, divisions=None, \*\*kwargs*)

Set the DataFrame index (row labels) using an existing column

This realigns the dataset to be sorted by a new column. This can have a significant impact on performance, because joins, groupbys, lookups, etc. are all much faster on that column. However, this performance increase comes with a cost, sorting a parallel dataset requires expensive shuffles. Often we `set_index` once directly after data ingest and filtering and then perform many cheap computations off of the sorted dataset.

This function operates exactly like `pandas.set_index` except with different performance costs (it is much more expensive). Under normal operation this function does an initial pass over the index column to compute approximate quantiles to serve as future divisions. It then passes over the data a second time, splitting up each input partition into several pieces and sharing those pieces to all of the output partitions now in sorted order.

In some cases we can alleviate those costs, for example if your dataset is sorted already then we can avoid making many small pieces or if you know good values to split the new index column then we can avoid the initial pass over the data. For example if your new index is a datetime index and your data is already sorted by day then this entire operation can be done for free. You can control these options with the following parameters.

### Parameters `df`: Dask DataFrame

**index**: string or Dask Series

**npartitions**: int, None, or 'auto'

The ideal number of output partitions. If None use the same as the input. If 'auto' then decide by memory use.

**shuffle**: string, optional

Either 'disk' for single-node operation or 'tasks' for distributed operation. Will be inferred by your current scheduler.

**sorted**: bool, optional

If the index column is already sorted in increasing order. Defaults to False

**divisions**: list, optional

Known values on which to separate index values of the partitions. See <http://dask.pydata.org/en/latest/dataframe-design.html#partitions> Defaults to computing this with a single pass over the data. Note that if `sorted=True`, specified divisions are assumed to match the existing partitions in the data. If this is untrue, you should leave divisions empty and call `repartition` after `set_index`.

**compute**: bool

Whether or not to trigger an immediate computation. Defaults to False.

## Examples

```
>>> df2 = df.set_index('x')
>>> df2 = df.set_index(d.x)
>>> df2 = df.set_index(d.timestamp, sorted=True)
```

A common case is when we have a datetime column that we know to be sorted and is cleanly divided by day. We can set this index for free by specifying both that the column is pre-sorted and the particular divisions along which it is separated

```
>>> import pandas as pd
>>> divisions = pd.date_range('2000', '2010', freq='1D')
>>> df2 = df.set_index('timestamp', sorted=True, divisions=divisions)
```

**shift** (*periods=1, freq=None, axis=0*)

Shift index by desired number of periods with an optional time freq

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, optional

Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

**axis** : {0 or 'index', 1 or 'columns'}

**Returns** **shifted** : DataFrame

## Notes

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

**size**

Size of the series

**std** (*axis=None, skipna=True, ddof=1, split\_every=False*)

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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** `std` : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the 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, *split\_every*=False)

Return the sum of the values for the requested axis

**Parameters** `axis` : {index (0), columns (1)}

`skipna` : boolean, default True

Exclude NA/null values when computing the result.

`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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

`min_count` : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 1. This means the sum or product of an all-NA or empty series is NaN.

**Returns** `sum` : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- `level`
- `numeric_only`
- `min_count`

## Examples

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([]).sum() # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

**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*)

Create Dask Bag from a Dask DataFrame

**Parameters** `index` : bool, optional

If True, the elements are tuples of (`index`, `value`), otherwise they're just the `value`. Default is False.

## Examples

```
>>> bag = df.to_bag()
```

**to\_csv** (*filename*, *\*\*kwargs*)

Store Dask DataFrame to CSV files

One filename per partition will be created. You can specify the filenames in a variety of ways.

Use a globstring:

```
>>> df.to_csv('/path/to/data/export-*.csv')
```

The \* will be replaced by the increasing sequence 0, 1, 2, ...

```
/path/to/data/export-0.csv  
/path/to/data/export-1.csv
```

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'
```

```
>>> df.to_csv('/path/to/data/export-*.csv', name_function=name)
```

```
/path/to/data/export-2015-01-01.csv  
/path/to/data/export-2015-01-02.csv  
...
```

You can also provide an explicit list of paths:

```
>>> paths = ['/path/to/data/alice.csv', '/path/to/data/bob.csv', ...]  
>>> df.to_csv(paths)
```

**Parameters** `filename` : string

Path glob indicating the naming scheme for the output files

**name\_function** : callable, default None

Function accepting an integer (partition index) and producing a string to replace the asterisk in the given filename globstring. Should preserve the lexicographic order of partitions

**compression** : string or None

String like 'gzip' or 'xz'. Must support efficient random access. Filenames with extensions corresponding to known compression algorithms (gz, bz2) will be compressed accordingly automatically

**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

**chunksiz**e : 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

**storage\_options**: dict

Parameters passed on to the backend filesystem class.

**Returns**

\_\_\_\_\_

**The names of the file written if they were computed right away**

**If not, the delayed tasks associated to the writing of the files**

**to\_delayed()**

Create Dask Delayed objects from a Dask Dataframe

Returns a list of delayed values, one value per partition.

### Examples

```
>>> partitions = df.to_delayed()
```

**to\_hdf** (*path\_or\_buf*, *key*, *mode='a'*, *append=False*, *get=None*, *\*\*kwargs*)

Store Dask Dataframe to Hierarchical Data Format (HDF) files

This is a parallel version of the Pandas function of the same name. Please see the Pandas docstring for more detailed information about shared keyword arguments.

This function differs from the Pandas version by saving the many partitions of a Dask DataFrame in parallel, either to many files, or to many datasets within the same file. You may specify this parallelism with an asterisk `*` within the filename or datapath, and an optional `name_function`. The asterisk will be replaced with an increasing sequence of integers starting from 0 or with the result of calling `name_function` on each of those integers.

This function only supports the Pandas `'table'` format, not the more specialized `'fixed'` format.

**Parameters path**: string

Path to a target filename. May contain a `*` to denote many filenames

**key**: string

Datapath within the files. May contain a `*` to denote many locations

**name\_function**: function

A function to convert the `*` in the above options to a string. Should take in a number from 0 to the number of partitions and return a string. (see examples below)

**compute**: bool

Whether or not to execute immediately. If `False` then this returns a `dask.Delayed` value.

**lock: Lock, optional**

Lock to use to prevent concurrency issues. By default a `threading.Lock`, `multiprocessing.Lock` or `SerializableLock` will be used depending on your scheduler if a lock is required. See `dask.utils.get_scheduler_lock` for more information about lock selection.

**\*\*other:**

See `pandas.to_hdf` for more information

**Returns** None: if `compute == True`

delayed value: if `compute == False`

**See also:**

`read_hdf`, `to_parquet`

**Examples**

Save Data to a single file

```
>>> df.to_hdf('output.hdf', '/data')
```

Save data to multiple datapaths within the same file:

```
>>> df.to_hdf('output.hdf', '/data-*')
```

Save data to multiple files:

```
>>> df.to_hdf('output-*.hdf', '/data')
```

Save data to multiple files, using the multiprocessing scheduler:

```
>>> df.to_hdf('output-*.hdf', '/data', get=dask.multiprocessing.get)
```

Specify custom naming scheme. This writes files as '2000-01-01.hdf', '2000-01-02.hdf', '2000-01-03.hdf', etc..

```
>>> from datetime import date, timedelta
>>> base = date(year=2000, month=1, day=1)
>>> def name_function(i):
...     ''' Convert integer 0 to n to a string '''
...     return base + timedelta(days=i)
```

```
>>> df.to_hdf('*.hdf', '/data', name_function=name_function)
```

**to\_html** (*max\_rows=5*)

Render a DataFrame as an HTML table.

*to\_html*-specific options:

**bold\_rows** [boolean, default True] Make the row labels bold in the output

**classes** [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

**escape** [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.=

**max\_rows** [int, optional] Maximum number of rows to show before truncating. If None, show all.

**max\_cols** [int, optional] Maximum number of columns to show before truncating. If None, show all.

**decimal** [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

**border** [int] A `border=border` attribute is included in the opening `<table>` tag. Default `pd.options.html.border`.

New in version 0.19.0.

**Parameters** **buf** : StringIO-like, optional

buffer to write to

**columns** : sequence, optional

the subset of columns to write; default None writes all columns

**col\_space** : int, optional

the minimum width of each column

**header** : bool, optional

whether to print column labels, default True

**index** : bool, optional

whether to print index (row) labels, default True

**na\_rep** : string, optional

string representation of NAN to use, default 'NaN'

**formatters** : list or dict of one-parameter functions, optional

formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

**float\_format** : one-parameter function, optional

formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

**sparsify** : bool, optional

Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

**index\_names** : bool, optional

Prints the names of the indexes, default True

**line\_width** : int, optional

Width to wrap a line in characters, default no wrap

**justify** : {'left', 'right', 'center', 'justify',

'justify-all', 'start', 'end', 'inherit', 'match-parent', 'initial', 'unset'}, default None

How to justify the column labels. If None uses the option from the print configuration (controlled by `set_option`), 'right' out of the box.

**Returns** **formatted** : string (or unicode, depending on data and options)

**to\_timestamp** (*freq=None, how='start', axis=0*)

Cast to DatetimeIndex of timestamps, at *beginning* of period

**Parameters** **freq** : string, default frequency of PeriodIndex

Desired frequency

**how** : {'s', 'e', 'start', 'end'}

Convention for converting period to timestamp; start of period vs. end

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

**copy** : boolean, default True

If false then underlying input data is not copied

**Returns** **df** : DataFrame with DatetimeIndex

## Notes

Dask doesn't support the following argument(s).

- copy

**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

## values

Return a `dask.array` of the values of this dataframe

Warning: This creates a `dask.array` without precise shape information. Operations that depend on shape information, like slicing or reshaping, will not work.

**var** (*axis=None, skipna=True, ddof=1, split\_every=False*)

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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns var** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the 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.display.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-like, or callable

Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. 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

Entries where *cond* is False are replaced with corresponding value from *other*. 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

**errors** : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**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)

Deprecated since version 0.21.0.

**Returns** **wh** : same type as caller

**See also:**

`DataFrame.mask()`

## Notes

The `where` method is an application of the if-then idiom. For each element in the calling `DataFrame`, if `cond` is `True` the element is used; otherwise the corresponding element from the `DataFrame` `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in [indexing](#).

## Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
```

```
>>> s.mask(s > 0)
0     0.0
1     NaN
2     NaN
3     NaN
4     NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A  B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
   A  B
0  True  True
1  True  True
2  True  True
```

```
3 True True
4 True True
```

## Series Methods

**class** `dask.dataframe.Series` (*dsk, name, meta, divisions*)  
Parallel Pandas Series

Do not use this class directly. Instead use functions like `dd.read_csv`, `dd.read_parquet`, or `dd.from_pandas`.

**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

**meta**: `pandas.Series`

An empty `pandas.Series` with names, dtypes, and index matching the expected output.

**divisions**: tuple of index values

Values along which we partition our blocks on the index

**See also:**

`dask.dataframe.DataFrame`

**abs** ()

Return an object with absolute value taken—only applicable to objects that are all numeric.

**Returns** `abs`: type of caller

**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`

**align** (*other, join='outer', axis=None, fill\_value=None*)

Align two objects on their axes with the specified join method for each axis Index

**Parameters other** : DataFrame or Series

**join** : { 'outer', 'inner', 'left', 'right' }, default 'outer'

**axis** : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

**level** : int or level name, default None

Broadcast across a level, matching Index values on the passed MultiIndex level

**copy** : boolean, default True

Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

**fill\_value** : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any "compatible" value

**method** : str, default None

**limit** : int, default None

**fill\_axis** : {0, 'index'}, default 0

Filling axis, method and limit

**broadcast\_axis** : {0, 'index'}, default None

Broadcast values along this axis, if aligning two objects of different dimensions

New in version 0.17.0.

**Returns (left, right)** : (Series, type of other)

Aligned objects

## Notes

Dask doesn't support the following argument(s).

- level
- copy
- method
- limit
- fill\_axis
- broadcast\_axis

**all** (*axis=None, skipna=True, split\_every=False*)

Return whether all elements are True 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

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns all** : Series or DataFrame (if level specified)

### Notes

Dask doesn't support the following argument(s).

- bool\_only
- level

**any** (*axis=None, skipna=True, split\_every=False*)

Return whether any element is True 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

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**Returns any** : Series or DataFrame (if level specified)

### Notes

Dask doesn't support the following argument(s).

- bool\_only
- level

**append** (*other*)

Concatenate two or more Series.

**Parameters to\_append** : Series or list/tuple of Series

**ignore\_index** : boolean, default False

If True, do not use the index labels.

**verify\_integrity** : boolean, default False

If True, raise Exception on creating index with duplicates

**Returns appended** : Series

**See also:**

`pandas.concat` General function to concatenate DataFrame, Series or Panel objects

## Notes

Iteratively appending to a Series can be more computationally intensive than a single concatenate. A better solution is to append values to a list and then concatenate the list with the original Series all at once.

## 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 `ignore_index` set to True:

```
>>> s1.append(s2, ignore_index=True)
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)
Traceback (most recent call last):
...
ValueError: Indexes have overlapping values: [0, 1, 2]
```

**apply** (*func*, *convert\_dtype=True*, *meta='\_\_no\_default\_\_'*, *args=()*, *\*\*kws*)

Parallel version of `pandas.Series.apply`

**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.

**meta** : pd.DataFrame, pd.Series, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**args** : tuple

Positional arguments to pass to function in addition to the value.

**Additional keyword arguments will be passed as keywords to the function.**

**Returns applied** : Series or DataFrame if func returns a Series.

**See also:**

`dask.Series.map_partitions`

## Examples

```
>>> import dask.dataframe as dd
>>> s = pd.Series(range(5), name='x')
>>> ds = dd.from_pandas(s, npartitions=2)
```

Apply a function elementwise across the Series, passing in extra arguments in `args` and `kwargs`:

```
>>> def myadd(x, a, b=1):
...     return x + a + b
>>> res = ds.apply(myadd, args=(2,), b=1.5)
```

By default, dask tries to infer the output metadata by running your provided function on some fake data. This works well in many cases, but can sometimes be expensive, or even fail. To avoid this, you can manually specify the output metadata with the `meta` keyword. This can be specified in many forms, for more information see `dask.dataframe.utils.make_meta`.

Here we specify the output is a Series with name 'x', and dtype float64:

```
>>> res = ds.apply(myadd, args=(2,), b=1.5, meta=('x', 'f8'))
```

In the case where the metadata doesn't change, you can also pass in the object itself directly:

```
>>> res = ds.apply(lambda x: x + 1, meta=ds)
```

**astype** (*dtype*)

Cast a pandas object to a specified dtype `dtype`.

**Parameters dtype** : data type, or dict of column name -> data type

Use a `numpy.dtype` or Python type to cast entire pandas object to the same type. Alternatively, use `{col: dtype, ...}`, where `col` is a column label and `dtype` is a `numpy.dtype` or Python type to cast one or more of the DataFrame's columns to column-specific types.

**copy** : bool, default True.

Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

**errors** : {'raise', 'ignore'}, default 'raise'.

Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

**raise\_on\_error** : raise on invalid input

Deprecated since version 0.20.0: Use `errors` instead

**kwargs** : keyword arguments to pass on to the constructor

**Returns** `casted` : type of caller

**See also:**

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Convert argument to a numeric type.

`numpy.ndarray.astype` Cast a numpy array to a specified type.

## Notes

Dask doesn't support the following argument(s).

- `copy`
- `errors`

## Examples

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int', copy=False)
>>> s2[0] = 10
>>> s1 # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

**autocorr** (*lag=1, split\_every=False*)

Lag-N autocorrelation

**Parameters lag** : int, default 1

Number of lags to apply before performing autocorrelation.

**Returns autocorr** : float

**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

**bfill** (*axis=None, limit=None*)

Synonym for `DataFrame.fillna(method='bfill')`

## Notes

Dask doesn't support the following argument(s).

- inplace
- downcast

**clear\_divisions** ()

Forget division information

**clip** (*lower=None, upper=None, out=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.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns clipped** : Series

## Notes

Dask doesn't support the following argument(s).

- axis
- inplace

## 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
```

**clip\_lower** (*threshold*)

Return copy of the input with values below given value(s) truncated.

**Parameters threshold** : float or array\_like

**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns clipped** : same type as input

**See also:**

*clip*

### Notes

Dask doesn't support the following argument(s).

- axis
- inplace

**clip\_upper** (*threshold*)

Return copy of input with values above given value(s) truncated.

**Parameters threshold** : float or array\_like

**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

**Whether to perform the operation in place on the data** New in version 0.21.0.

**Returns clipped** : same type as input

**See also:**

*clip*

### Notes

Dask doesn't support the following argument(s).

- axis
- inplace

**combine** (*other, func, fill\_value=None*)

Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other

**Parameters other** : Series or scalar value

**func** : function

Function that takes two scalars as inputs and return a scalar

**fill\_value** : scalar value

**Returns result** : Series

**See also:**

*Series.combine\_first* Combine Series values, choosing the calling Series's values first

### Examples

```
>>> s1 = Series([1, 2])
>>> s2 = Series([0, 3])
>>> s1.combine(s2, lambda x1, x2: x1 if x1 < x2 else x2)
0    0
1    2
dtype: int64
```

#### **combine\_first** (*other*)

Combine Series values, choosing the calling Series's values first. Result index will be the union of the two indexes

**Parameters** *other* : Series

**Returns** *combined* : Series

**See also:**

*Series.combine* Perform elementwise operation on two Series using a given function

### Examples

```
>>> s1 = pd.Series([1, np.nan])
>>> s2 = pd.Series([3, 4])
>>> s1.combine_first(s2)
0    1.0
1    4.0
dtype: float64
```

#### **compute** (\*\**kwargs*)

Compute this dask collection

This turns a lazy Dask collection into its in-memory equivalent. For example a `Dask.array` turns into a NumPy array and a `Dask.dataframe` turns into a Pandas dataframe. The entire dataset must fit into memory before calling this operation.

**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.

**See also:**

`dask.base.compute`

**copy** ()

Make a copy of the dataframe

This is strictly a shallow copy of the underlying computational graph. It does not affect the underlying data

**corr** (*other*, *method*='pearson', *min\_periods*=None, *split\_every*=False)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** (*split\_every*=False)

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 support the following argument(s).

- level

**cov** (*other*, *min\_periods*=None, *split\_every*=False)

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 max 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**See also:**

`pandas.core.window.Expanding.max` Similar functionality but ignores NaN values.

**cummin** (*axis=None, skipna=True*)

Return cumulative minimum 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

**See also:**

`pandas.core.window.Expanding.min` Similar functionality but ignores NaN values.

**cumprod** (*axis=None, skipna=True*)

Return cumulative product 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

**See also:**

`pandas.core.window.Expanding.prod` Similar functionality but ignores NaN values.

**cumsum** (*axis=None, skipna=True*)

Return cumulative sum 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

**See also:**

`pandas.core.window.Expanding.sum` Similar functionality but ignores NaN values.

**describe** (*split\_every=False*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters percentiles** : list-like of numbers, optional

The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

**include** : 'all', list-like of dtypes or None (default), optional

A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use 'category'
- None (default) : The result will include all numeric columns.

**exclude** : list-like of dtypes or None (default), optional,

A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use 'category'
- None (default) : The result will exclude nothing.

**Returns** summary: Series/DataFrame of summary statistics

**See also:**

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

## Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as `lower`, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

## Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
```

```
std      1.0
min      1.0
25%     1.5
50%     2.0
75%     2.5
max      3.0
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count      4
unique     3
top        a
freq       2
dtype: object
```

Describing a timestamp Series.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                3
unique               2
top      2010-01-01 00:00:00
freq                2
first    2000-01-01 00:00:00
last     2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'object': ['a', 'b', 'c'],
...                    'numeric': [1, 2, 3],
...                    'categorical': pd.Categorical(['d', 'e', 'f'])
...                    })
>>> df.describe()
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%       1.5
50%       2.0
75%       2.5
max        3.0
```

Describing all columns of a DataFrame regardless of data type.

```
>>> df.describe(include='all')
      categorical  numeric  object
count           3        3.0      3
unique          3        NaN      3
top             f        NaN      c
freq           1        NaN      1
```

mean	NaN	2.0	NaN
std	NaN	1.0	NaN
min	NaN	1.0	NaN
25%	NaN	1.5	NaN
50%	NaN	2.0	NaN
75%	NaN	2.5	NaN
max	NaN	3.0	NaN

Describing a column from a DataFrame by accessing it as an attribute.

```
>>> df.numeric.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%       1.5
50%       2.0
75%       2.5
max        3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a DataFrame description.

```
>>> df.describe(include=[np.number])
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%       1.5
50%       2.0
75%       2.5
max        3.0
```

Including only string columns in a DataFrame description.

```
>>> df.describe(include=[np.object])
      object
count      3
unique     3
top        c
freq       1
```

Including only categorical columns from a DataFrame description.

```
>>> df.describe(include=['category'])
      categorical
count          3
unique         3
top            f
freq           1
```

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
      categorical object
count          3      3
unique         3      3
```

top	f	c
freq	1	1

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
           categorical  numeric
count              3         3.0
unique             3         NaN
top               f         NaN
freq              1         NaN
mean             NaN         2.0
std              NaN         1.0
min              NaN         1.0
25%             NaN         1.5
50%             NaN         2.0
75%             NaN         2.5
max              NaN         3.0
```

**diff** (*periods=1, axis=0*)

1st discrete difference of object

**Parameters** **periods** : int, default 1

Periods to shift for forming difference

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Take difference over rows (0) or columns (1).

**Returns** **dified** : DataFrame

**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** (*split\_every=None, split\_out=1, \*\*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.

**inplace** : boolean, default False

Whether to drop duplicates in place or to return a copy

**Returns deduplicated** : DataFrame

### Notes

Dask doesn't support the following argument(s).

- `subset`
- `keep`
- `inplace`

### **dropna** ()

Return Series without null values

**Returns valid** : Series

**inplace** : boolean, default False

Do operation in place.

### Notes

Dask doesn't support the following argument(s).

- `axis`
- `inplace`

### **dtype**

Return data type

### **eq** (*other*, *level=None*, *axis=0*)

Equal to of series and other, element-wise (binary operator *eq*).

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.None`

**ffill** (*axis=None, limit=None*)

Synonym for `DataFrame.fillna(method='ffill')`

## Notes

Dask doesn't support the following argument(s).

- inplace
- downcast

**fillna** (*value=None, method=None, limit=None, axis=None*)

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 or 'index', 1 or 'columns'}

**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. Must be greater than 0 if not None.

**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** : DataFrame

**See also:**

`reindex`, `asfreq`

## Notes

Dask doesn't support the following argument(s).

- inplace
- downcast

## Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
   A    B    C    D
0 NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2 NaN  NaN NaN  5
3 NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
   A    B    C    D
0  0.0  2.0  0.0  0
1  3.0  4.0  0.0  1
2  0.0  0.0  0.0  5
3  0.0  3.0  0.0  4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
   A    B    C    D
0 NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  3.0  4.0 NaN  5
3  3.0  3.0 NaN  4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  2.0  1
2  0.0  1.0  2.0  5
3  0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0 NaN  1
2  NaN  1.0 NaN  5
3  NaN  3.0 NaN  4
```

### **first** (*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

**Parameters** *offset*: string, DateOffset, dateutil.relativedelta

**Returns** *subset*: type of caller

## Examples

`ts.first('10D')` -> First 10 days

**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`

**ge** (*other*, *level=None*, *axis=0*)

Greater than or equal to of series and other, element-wise (binary operator *ge*).

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.None`

**get\_partition** (*n*)

Get a dask DataFrame/Series representing the *nth* partition.

**groupby** (*by=None*, *\*\*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, str, or iterable

Used to determine the groups for the groupby. If *by* is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A str or list of str may be passed to group by the columns in `self`

**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 support the following argument(s).

- 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()
```

**gt** (*other*, *level=None*, *axis=0*)

Greater than of series and other, element-wise (binary operator *gt*).

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.None`

**head** (*n=5, npartitions=1, compute=True*)

First n rows of the dataset

**Parameters n** : int, optional

The number of rows to return. Default is 5.

**npartitions** : int, optional

Elements are only taken from the first `npartitions`, with a default of 1. If there are fewer than `n` rows in the first `npartitions` a warning will be raised and any found rows returned. Pass -1 to use all partitions.

**compute** : bool, optional

Whether to compute the result, default is True.

**idxmax** (*axis=None, skipna=True, split\_every=False*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

**Parameters axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns idxmax** : Series

**Raises ValueError**

- If the row/column is empty

**See also:**

`Series.idxmax`

## Notes

This method is the DataFrame version of `ndarray.argmax`.

**idxmin** (*axis=None, skipna=True, split\_every=False*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

**Parameters axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns** `idxmin` : Series

**Raises** `ValueError`

- If the row/column is empty

**See also:**

`Series.idxmin`

## Notes

This method is the DataFrame version of `ndarray.argmax`.

### index

Return dask Index instance

### `isin` (*values*)

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`

## 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 NA.

**See also:**

**DataFrame.notna** boolean inverse of isna

**DataFrame.isnull** alias of isna

**isna** top-level isna

**iteritems** ()

Lazily iterate over (index, value) tuples

**known\_divisions**

Whether divisions are already known

**last** (*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

**Parameters** **offset** : string, DateOffset, dateutil.relativedelta

**Returns** **subset** : type of caller

**Examples**

```
ts.last('5M') -> Last 5 months
```

**le** (*other, level=None, axis=0*)

Less than or equal to of series and other, element-wise (binary operator *le*).

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.None`

**loc**

Purely label-location based indexer for selection by label.

```
>>> df.loc["b"]
>>> df.loc["b":"d"]
```

**lt** (*other, level=None, axis=0*)

Less than of series and other, element-wise (binary operator *lt*).

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.None`

**map** (*arg*, *na\_action=None*, *meta='\_\_no\_default\_\_'*)

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, without passing them to the mapping function

**meta** : `pd.DataFrame`, `pd.Series`, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of {name: dtype} or iterable of (name, dtype) can be provided. Instead of a series, a tuple of (name, dtype) can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**Returns y** : Series

same index as caller

**See also:**

[`Series.apply`](#) For applying more complex functions on a Series

[`DataFrame.apply`](#) Apply a function row-/column-wise

[`DataFrame.applymap`](#) Apply a function elementwise on a whole DataFrame

## Notes

When *arg* is a dictionary, values in Series that are not in the dictionary (as keys) are converted to NaN. However, if the dictionary is a dict subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than NaN:

```
>>> from collections import Counter
>>> counter = Counter()
>>> counter['bar'] += 1
>>> y.map(counter)
1    0
2    1
3    0
dtype: int64
```

## Examples

Map inputs to outputs (both of type *Series*)

```
>>> x = pd.Series([1,2,3], index=['one', 'two', 'three'])
>>> x
one      1
two      2
three    3
dtype: int64
```

```
>>> y = pd.Series(['foo', 'bar', 'baz'], index=[1,2,3])
>>> y
1      foo
2      bar
3      baz
```

```
>>> x.map(y)
one      foo
two      bar
three    baz
```

If *arg* is a dictionary, return a new Series with values converted according to the dictionary's mapping:

```
>>> z = {1: 'A', 2: 'B', 3: 'C'}
```

```
>>> x.map(z)
one      A
two      B
three    C
```

Use *na\_action* to control whether NA values are affected by the mapping function.

```
>>> s = pd.Series([1, 2, 3, np.nan])
```

```
>>> s2 = s.map('this is a string {}'.format, na_action=None)
0      this is a string 1.0
1      this is a string 2.0
2      this is a string 3.0
3      this is a string nan
dtype: object
```

```
>>> s3 = s.map('this is a string {}'.format, na_action='ignore')
0      this is a string 1.0
1      this is a string 2.0
2      this is a string 3.0
3                                     NaN
dtype: object
```

**map\_overlap** (*func, before, after, \*args, \*\*kwargs*)

Apply a function to each partition, sharing rows with adjacent partitions.

This can be useful for implementing windowing functions such as `df.rolling(...).mean()` or `df.diff()`.

**Parameters `func` :** function

Function applied to each partition.

**before :** int

The number of rows to prepend to partition  $i$  from the end of partition  $i - 1$ .

**after :** int

The number of rows to append to partition  $i$  from the beginning of partition  $i + 1$ .

**args, kwargs :**

Arguments and keywords to pass to the function. The partition will be the first argument, and these will be passed *after*.

**meta :** `pd.DataFrame`, `pd.Series`, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**Notes**

Given positive integers `before` and `after`, and a function `func`, `map_overlap` does the following:

1. Prepend `before` rows to each partition  $i$  from the end of partition  $i - 1$ . The first partition has no rows prepended.
2. Append `after` rows to each partition  $i$  from the beginning of partition  $i + 1$ . The last partition has no rows appended.
3. Apply `func` to each partition, passing in any extra `args` and `kwargs` if provided.
4. Trim `before` rows from the beginning of all but the first partition.
5. Trim `after` rows from the end of all but the last partition.

Note that the index and divisions are assumed to remain unchanged.

**Examples**

Given a `DataFrame`, `Series`, or `Index`, such as:

```
>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': [1, 2, 4, 7, 11],
...                   'y': [1., 2., 3., 4., 5.]})
>>> ddf = dd.from_pandas(df, npartitions=2)
```

A rolling sum with a trailing moving window of size 2 can be computed by overlapping 2 rows before each partition, and then mapping calls to `df.rolling(2).sum()`:

```

>>> ddf.compute()
   x   y
0  1  1.0
1  2  2.0
2  4  3.0
3  7  4.0
4 11  5.0
>>> ddf.map_overlap(lambda df: df.rolling(2).sum(), 2, 0).compute()
   x   y
0  NaN NaN
1  3.0 3.0
2  6.0 5.0
3 11.0 7.0
4 18.0 9.0

```

The pandas `diff` method computes a discrete difference shifted by a number of periods (can be positive or negative). This can be implemented by mapping calls to `df.diff` to each partition after prepending/appending that many rows, depending on sign:

```

>>> def diff(df, periods=1):
...     before, after = (periods, 0) if periods > 0 else (0, -periods)
...     return df.map_overlap(lambda df: df.diff(periods),
...                           periods, 0, periods=periods)
>>> diff(ddf, 1).compute()
   x   y
0  NaN NaN
1  1.0 1.0
2  2.0 1.0
3  3.0 1.0
4  4.0 1.0

```

If you have a `DatetimeIndex`, you can use a `timedelta` for time-based windows. `>>> ts = pd.Series(range(10), index=pd.date_range('2017', periods=10)) >>> dts = dd.from_pandas(ts, npartitions=2) >>> dts.map_overlap(lambda df: df.rolling('2D').sum(), ... pd.Timedelta('2D'), 0).compute()`  
2017-01-01 0.0 2017-01-02 1.0 2017-01-03 3.0 2017-01-04 5.0 2017-01-05 7.0 2017-01-06 9.0 2017-01-07 11.0 2017-01-08 13.0 2017-01-09 15.0 2017-01-10 17.0 dtype: float64

**map\_partitions** (*func*, \**args*, \*\**kwargs*)

Apply Python function on each DataFrame partition.

Note that the index and divisions are assumed to remain unchanged.

**Parameters func** : function

Function applied to each partition.

**args, kwargs** :

Arguments and keywords to pass to the function. The partition will be the first argument, and these will be passed *after*.

**meta** : `pd.DataFrame`, `pd.Series`, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can be used. If not provided, dask will try to infer the metadata. This may lead to

unexpected results, so providing meta is recommended. For more information, see `dask.dataframe.utils.make_meta`.

## Examples

Given a DataFrame, Series, or Index, such as:

```
>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': [1, 2, 3, 4, 5],
...                   'y': [1., 2., 3., 4., 5.]})
>>> ddf = dd.from_pandas(df, npartitions=2)
```

One can use `map_partitions` to apply a function on each partition. Extra arguments and keywords can optionally be provided, and will be passed to the function after the partition.

Here we apply a function with arguments and keywords to a DataFrame, resulting in a Series:

```
>>> def myadd(df, a, b=1):
...     return df.x + df.y + a + b
>>> res = ddf.map_partitions(myadd, 1, b=2)
>>> res.dtype
dtype('float64')
```

By default, dask tries to infer the output metadata by running your provided function on some fake data. This works well in many cases, but can sometimes be expensive, or even fail. To avoid this, you can manually specify the output metadata with the `meta` keyword. This can be specified in many forms, for more information see `dask.dataframe.utils.make_meta`.

Here we specify the output is a Series with no name, and dtype `float64`:

```
>>> res = ddf.map_partitions(myadd, 1, b=2, meta=(None, 'f8'))
```

Here we map a function that takes in a DataFrame, and returns a DataFrame with a new column:

```
>>> res = ddf.map_partitions(lambda df: df.assign(z=df.x * df.y))
>>> res.dtypes
x      int64
y      float64
z      float64
dtype: object
```

As before, the output metadata can also be specified manually. This time we pass in a dict, as the output is a DataFrame:

```
>>> res = ddf.map_partitions(lambda df: df.assign(z=df.x * df.y),
...                          meta={'x': 'i8', 'y': 'f8', 'z': 'f8'})
```

In the case where the metadata doesn't change, you can also pass in the object itself directly:

```
>>> res = ddf.map_partitions(lambda df: df.head(), meta=df)
```

Also note that the index and divisions are assumed to remain unchanged. If the function you're mapping changes the index/divisions, you'll need to clear them afterwards:

```
>>> ddf.map_partitions(func).clear_divisions()
```

**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-like, or callable

Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. 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

Entries where *cond* is True are replaced with corresponding value from *other*. 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

**errors** : str, {'raise', 'ignore'}, default 'raise'

- *raise* : allow exceptions to be raised
- *ignore* : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**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)

Deprecated since version 0.21.0.

**Returns** **wh** : same type as caller

**See also:**

*DataFrame.where()*

## Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if *cond* is False the element is used; otherwise the corresponding element from the DataFrame *other* is used.

The signature for *DataFrame.where()* differs from *numpy.where()*. Roughly *df1.where(m, df2)* is equivalent to *np.where(m, df1, df2)*.

For further details and examples see the *mask* documentation in [indexing](#).

## Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A    B
0  True True
1  True True
2  True True
3  True True
4  True True
>>> df.where(m, -df) == df.mask(~m, -df)
   A    B
0  True True
1  True True
2  True True
3  True True
4  True True
```

**max** (*axis=None, skipna=True, split\_every=False*)

**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 when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** **max** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- level
- numeric\_only

**mean** (*axis=None, skipna=True, split\_every=False*)

Return the mean of the values for the requested axis

**Parameters** **axis** : {index (0), columns (1)}

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** **mean** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- level
- numeric\_only

**memory\_usage** (*index=True, deep=False*)

Memory usage of the Series

**Parameters** **index** : bool

Specifies whether to include memory usage of Series index

**deep** : bool

Inspect the data deeply, interrogate *object* dtypes for system-level memory consumption

**Returns** scalar bytes of memory consumed

**See also:**`numpy.ndarray.nbytes`**Notes**

Memory usage does not include memory consumed by elements that are not components of the array if `deep=False`

**min** (*axis=None, skipna=True, split\_every=False*)

**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 when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** **min** : Series or DataFrame (if level specified)

**Notes**

Dask doesn't support the 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`

**nbytes**

Number of bytes

**ndim**

Return dimensionality

**ne** (*other*, *level=None*, *axis=0*)

Not equal to of series and other, element-wise (binary operator *ne*).

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.None`

**nlargest** (*n=5*, *split\_every=None*)

Return the largest *n* elements.

**Parameters n** : int

Return this many descending sorted values

**keep** : {'first', 'last'}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

**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(10**6))
>>> s.nlargest(10) # only sorts up to the N requested
219921    4.644710
82124    4.608745
421689    4.564644
425277    4.447014
718691    4.414137
43154    4.403520
283187    4.313922
595519    4.273635
503969    4.250236
121637    4.240952
dtype: float64
```

**notnull()**

Return a boolean same-sized object indicating if the values are not NA.

**See also:**

**DataFrame.isna** boolean inverse of notna

*DataFrame.notnull* alias of notna

**notna** top-level notna

**npartitions**

Return number of partitions

**nsmallest** ( $n=5$ , *split\_every=None*)

Return the smallest  $n$  elements.

**Parameters**  $n$  : int

Return this many ascending sorted values

**keep** : {'first', 'last'}, default 'first'

Where there are duplicate values: - *first* : take the first occurrence. - *last* : take the last occurrence.

**Returns** **bottom\_n** : Series

The  $n$  smallest values in the Series, in sorted order

**See also:***Series.nlargest*

## Notes

Faster than `.sort_values().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(10**6))
>>> s.nsmallest(10) # only sorts up to the N requested
288532    -4.954580
732345    -4.835960
64803     -4.812550
446457    -4.609998
501225    -4.483945
669476    -4.472935
973615    -4.401699
621279    -4.355126
773916    -4.347355
359919    -4.331927
dtype: float64
```

### **nunique** (*split\_every=None*)

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 support the following argument(s).

- `dropna`

### **nunique\_approx** (*split\_every=None*)

Approximate number of unique rows.

This method uses the HyperLogLog algorithm for cardinality estimation to compute the approximate number of unique rows. The approximate error is 0.406%.

**Parameters** `split_every` : int, optional

Group partitions into groups of this size while performing a tree-reduction. If set to False, no tree-reduction will be used. Default is 8.

**Returns** a float representing the approximate number of elements

### **persist** (*\*\*kwargs*)

Persist this dask collection into memory

This turns a lazy Dask collection into a Dask collection with the same metadata, but now with the results fully computed or actively computing in the background.

**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.

**Returns** New dask collections backed by in-memory data

**See also:**

`dask.base.persist`

**pipe** (*func*, \*args, \*\*kwargs)

Apply `func(self, *args, **kwargs)`

**Parameters** **func** : function

function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a (`callable`, `data_keyword`) tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

**args** : iterable, optional

positional arguments passed into `func`.

**kwargs** : mapping, optional

a dictionary of keyword arguments passed into `func`.

**Returns** **object** : the return type of `func`.

**See also:**

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

## Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe((f, 'arg2'), arg1=a, arg3=c)
... )
```

**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*

**prod** (*axis=None*, *skipna=True*, *split\_every=False*)

Return the product of the values for the requested axis

**Parameters** *axis* : {index (0), columns (1)}

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than *min\_count* non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 1. This means the sum or product of an all-NA or empty series is NaN.

**Returns** *prod* : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- *level*
- *numeric\_only*
- *min\_count*

## Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

### **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** (*frac, random\_state=None*)

Pseudorandomly split dataframe into different pieces row-wise

**Parameters frac** : list

List of floats that should sum to one.

**random\_state**: int or `np.random.RandomState`

If int create a new `RandomState` with this as the seed

**Otherwise draw from the passed `RandomState`**

**See also:**

`dask.DataFrame.sample`

## 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*

**reduction** (*chunk*, *aggregate=None*, *combine=None*, *meta='\_\_no\_default\_\_'*, *token=None*, *split\_every=None*, *chunk\_kwargs=None*, *aggregate\_kwargs=None*, *combine\_kwargs=None*, *\*\*kwargs*)

Generic row-wise reductions.

**Parameters chunk** : callable

Function to operate on each partition. Should return a `pandas.DataFrame`, `pandas.Series`, or a scalar.

**aggregate** : callable, optional

Function to operate on the concatenated result of *chunk*. If not specified, defaults to *chunk*. Used to do the final aggregation in a tree reduction.

The input to *aggregate* depends on the output of *chunk*. If the output of *chunk* is a:

- scalar: Input is a Series, with one row per partition.
- Series: Input is a DataFrame, with one row per partition. Columns are the rows in the output series.
- DataFrame: Input is a DataFrame, with one row per partition. Columns are the columns in the output dataframes.

Should return a `pandas.DataFrame`, `pandas.Series`, or a scalar.

**combine** : callable, optional

Function to operate on intermediate concatenated results of `chunk` in a tree-reduction. If not provided, defaults to `aggregate`. The input/output requirements should match that of `aggregate` described above.

**meta** : `pd.DataFrame`, `pd.Series`, `dict`, `iterable`, `tuple`, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a `dict` of `{name: dtype}` or `iterable` of `(name, dtype)` can be provided. Instead of a `Series`, a `tuple` of `(name, dtype)` can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**token** : `str`, optional

The name to use for the output keys.

**split\_every** : `int`, optional

Group partitions into groups of this size while performing a tree-reduction. If set to `False`, no tree-reduction will be used, and all intermediates will be concatenated and passed to `aggregate`. Default is 8.

**chunk\_kwargs** : `dict`, optional

Keyword arguments to pass on to `chunk` only.

**aggregate\_kwargs** : `dict`, optional

Keyword arguments to pass on to `aggregate` only.

**combine\_kwargs** : `dict`, optional

Keyword arguments to pass on to `combine` only.

**kwargs** :

All remaining keywords will be passed to `chunk`, `combine`, and `aggregate`.

## Examples

```
>>> import pandas as pd
>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': range(50), 'y': range(50, 100)})
>>> ddf = dd.from_pandas(df, npartitions=4)
```

Count the number of rows in a `DataFrame`. To do this, count the number of rows in each partition, then sum the results:

```
>>> res = ddf.reduction(lambda x: x.count(),
...                    aggregate=lambda x: x.sum())
>>> res.compute()
x      50
y      50
dtype: int64
```

Count the number of rows in a `Series` with elements greater than or equal to a value (provided via a keyword).

```
>>> def count_greater(x, value=0):
...     return (x >= value).sum()
>>> res = ddf.x.reduction(count_greater, aggregate=lambda x: x.sum(),
...                       chunk_kwargs={'value': 25})
>>> res.compute()
25
```

Aggregate both the sum and count of a Series at the same time:

```
>>> def sum_and_count(x):
...     return pd.Series({'sum': x.sum(), 'count': x.count()})
>>> res = ddf.x.reduction(sum_and_count, aggregate=lambda x: x.sum())
>>> res.compute()
count      50
sum      1225
dtype: int64
```

Doing the same, but for a DataFrame. Here `chunk` returns a DataFrame, meaning the input to `aggregate` is a DataFrame with an index with non-unique entries for both 'x' and 'y'. We groupby the index, and sum each group to get the final result.

```
>>> def sum_and_count(x):
...     return pd.DataFrame({'sum': x.sum(), 'count': x.count()})
>>> res = ddf.reduction(sum_and_count,
...                     aggregate=lambda x: x.groupby(level=0).sum())
>>> res.compute()
count  sum
x      50 1225
y      50 3725
```

**rename** (*index=None, inplace=False, sorted\_index=False*)

Alter Series index labels or name

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

Alternatively, change `Series.name` with a scalar value.

**Parameters** **index** : scalar, hashable sequence, dict-like or callable, optional

If dict-like or callable, the transformation is applied to the index. Scalar or hashable sequence-like will alter the `Series.name` attribute.

**inplace** : boolean, default False

Whether to return a new Series or modify this one inplace.

**sorted\_index** : bool, default False

If true, the output Series will have known divisions inferred from the input series and the transformation. Ignored for non-callable/dict-like `index` or when the input series has unknown divisions. Note that this may only be set to `True` if you know that the transformed index is monotonically increasing. Dask will check that transformed divisions are monotonic, but cannot check all the values between divisions, so incorrectly setting this can result in bugs.

**Returns** **renamed** : Series

**See also:**

`pandas.Series.rename`

**repartition** (*divisions=None, npartitions=None, freq=None, force=False*)

Repartition dataframe along new divisions

**Parameters** **divisions** : list, optional

List of partitions to be used. If specified npartitions will be ignored.

**npartitions** : int, optional

Number of partitions of output, must be less than npartitions of input. Only used if divisions isn't specified.

**freq** : str, pd.Timedelta

A period on which to partition timeseries data like '7D' or '12h' or pd.Timedelta(hours=12). Assumes a datetime index.

**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])
>>> df = df.repartition(freq='7d')
```

**resample** (*rule, how=None, closed=None, label=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

**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. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**label** : {'right', 'left'}

Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**convention** : {'start', 'end', 's', 'e'}

For PeriodIndex only, controls whether to use the start or end of *rule*

**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

**on** : string, optional

For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

**level** : string or int, optional

For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

## Notes

To learn more about the offset strings, please see [this link](#).

## 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    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00    1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
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
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))

>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using *'start' convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using *'end' convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
      a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                      columns=['a', 'b', 'c', 'd'],
                      index=pd.MultiIndex.from_product([time, [1, 2]]))
>>> df2.resample('3T', level=0).sum()
      a  b  c  d
```

```
2000-01-01 00:00:00 0 6 12 18
2000-01-01 00:03:00 0 4 8 12
```

**reset\_index** (*drop=False*)

Reset the index to the default index.

Note that unlike in `pandas`, the reset `dask.dataframe` index will not be monotonically increasing from 0. Instead, it will restart at 0 for each partition (e.g. `index1 = [0, ..., 10]`, `index2 = [0, ...]`). This is due to the inability to statically know the full length of the index.

For `DataFrame` with multi-level index, returns a 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** `drop` : boolean, default `False`

Do not try to insert index into dataframe columns.

**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, str, offset

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. If using an offset or offset alias like '5D', the data must have a `DatetimeIndex`

Changed in version 0.15.0: Now accepts offsets and string offset aliases

**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.

**round** (*decimals=0*)

Round each value in a Series to the given number of decimals.

**Parameters** **decimals** : int

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.

**Returns** Series object

**See also:**

`numpy.around`, `DataFrame.round`

**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

**See also:**`DataFrame.random_split`, `pandas.DataFrame.sample`**sem** (*axis=None*, *skipna=None*, *ddof=1*, *split\_every=False*)

Return unbiased standard error of the mean 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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** **sem** : Series or DataFrame (if level specified)**Notes**

Dask doesn't support the following argument(s).

- level
- numeric\_only

**shift** (*periods=1*, *freq=None*, *axis=0*)

Shift index by desired number of periods with an optional time freq

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, optional

Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

**axis** : {0 or 'index', 1 or 'columns'}

**Returns shifted** : DataFrame

## Notes

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

### size

Size of the series

**std** (*axis=None, skipna=True, ddof=1, split\_every=False*)

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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns std** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the 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, split\_every=False*)

Return the sum of the values for the requested axis

**Parameters axis** : {index (0), columns (1)}

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 1. This means the sum or product of an all-NA or empty series is NaN.

**Returns sum** : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the following argument(s).

- level
- numeric\_only
- min\_count

## Examples

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([]).sum() # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

**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*)

Create a Dask Bag from a Series

**to\_csv** (*filename, \*\*kwargs*)

Store Dask DataFrame to CSV files

One filename per partition will be created. You can specify the filenames in a variety of ways.

Use a globstring:

```
>>> df.to_csv('/path/to/data/export-*.csv')
```

The *\** will be replaced by the increasing sequence 0, 1, 2, ...

```
/path/to/data/export-0.csv
/path/to/data/export-1.csv
```

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'
```

```
>>> df.to_csv('/path/to/data/export-*.csv', name_function=name)
```

```
/path/to/data/export-2015-01-01.csv
/path/to/data/export-2015-01-02.csv
...
```

You can also provide an explicit list of paths:

```
>>> paths = ['/path/to/data/alice.csv', '/path/to/data/bob.csv', ...]
>>> df.to_csv(paths)
```

**Parameters** `filename` : string

Path glob indicating the naming scheme for the output files

`name_function` : callable, default None

Function accepting an integer (partition index) and producing a string to replace the asterisk in the given filename globstring. Should preserve the lexicographic order of partitions

**compression** : string or None

String like 'gzip' or 'xz'. Must support efficient random access. Filenames with extensions corresponding to known compression algorithms (gz, bz2) will be compressed accordingly automatically

**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

**storage\_options**: dict  
 Parameters passed on to the backend filesystem class.

**Returns**

\_\_\_\_\_

**The names of the file written if they were computed right away**  
**If not, the delayed tasks associated to the writing of the files**

**to\_delayed()**  
 Create Dask Delayed objects from a Dask Dataframe  
 Returns a list of delayed values, one value per partition.

**Examples**

```
>>> partitions = df.to_delayed()
```

**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, get=None, \*\*kwargs*)  
 Store Dask Dataframe to Hierarchical Data Format (HDF) files

This is a parallel version of the Pandas function of the same name. Please see the Pandas docstring for more detailed information about shared keyword arguments.

This function differs from the Pandas version by saving the many partitions of a Dask DataFrame in parallel, either to many files, or to many datasets within the same file. You may specify this parallelism with an asterisk `*` within the filename or datapath, and an optional `name_function`. The asterisk will be replaced with an increasing sequence of integers starting from 0 or with the result of calling `name_function` on each of those integers.

This function only supports the Pandas `'table'` format, not the more specialized `'fixed'` format.

**Parameters** `path`: string

Path to a target filename. May contain a `*` to denote many filenames

**key**: string

Datapath within the files. May contain a `*` to denote many locations

**name\_function**: function

A function to convert the `*` in the above options to a string. Should take in a number from 0 to the number of partitions and return a string. (see examples below)

**compute**: bool

Whether or not to execute immediately. If False then this returns a `dask.Delayed` value.

**lock**: Lock, optional

Lock to use to prevent concurrency issues. By default a `threading.Lock`, `multiprocessing.Lock` or `SerializableLock` will be used depending on your scheduler if a lock is required. See `dask.utils.get_scheduler_lock` for more information about lock selection.

**\*\*other**:

See `pandas.to_hdf` for more information

**Returns** None: if `compute == True`

delayed value: if `compute == False`

**See also:**

`read_hdf`, `to_parquet`

## Examples

Save Data to a single file

```
>>> df.to_hdf('output.hdf', '/data')
```

Save data to multiple datapaths within the same file:

```
>>> df.to_hdf('output.hdf', '/data-*')
```

Save data to multiple files:

```
>>> df.to_hdf('output-*.hdf', '/data')
```

Save data to multiple files, using the multiprocessing scheduler:

```
>>> df.to_hdf('output-*.hdf', '/data', get=dask.multiprocessing.get)
```

Specify custom naming scheme. This writes files as '2000-01-01.hdf', '2000-01-02.hdf', '2000-01-03.hdf', etc..

```
>>> from datetime import date, timedelta
>>> base = date(year=2000, month=1, day=1)
>>> def name_function(i):
...     ''' Convert integer 0 to n to a string '''
...     return base + timedelta(days=i)
```

```
>>> df.to_hdf('*.hdf', '/data', name_function=name_function)
```

**to\_parquet** (*path*, \*args, \*\*kwargs)

Store Dask.dataframe to Parquet files

**Parameters** *df*: dask.dataframe.DataFrame

**path**: string

Destination directory for data. Prepend with protocol like `s3://` or `hdfs://` for remote data.

**engine**: {'auto', 'fastparquet', 'pyarrow'}, default 'auto'

Parquet library to use. If only one library is installed, it will use that one; if both, it will use 'fastparquet'.

**compression**: string or dict, optional

Either a string like "snappy" or a dictionary mapping column names to compressors like {"name": "gzip", "values": "snappy"}. The default is "default", which uses the default compression for whichever engine is selected.

**write\_index**: boolean, optional

Whether or not to write the index. Defaults to True *if* divisions are known.

**append**: bool, optional

If False (default), construct data-set from scratch. If True, add new row-group(s) to an existing data-set. In the latter case, the data-set must exist, and the schema must match the input data.

**ignore\_divisions**: bool, optional

If False (default) raises error when previous divisions overlap with the new appended divisions. Ignored if `append=False`.

**partition\_on**: list, optional

Construct directory-based partitioning by splitting on these fields' values. Each dask partition will result in one or more datafiles, there will be no global groupby.

**storage\_options**: dict, optional

Key/value pairs to be passed on to the file-system backend, if any.

**compute**: bool, optional

If True (default) then the result is computed immediately. If False then a `dask.delayed` object is returned for future computation.

**\*\*kwargs**

Extra options to be passed on to the specific backend.

**See also:**

[\*read\\_parquet\*](#) Read parquet data to dask.dataframe

**Notes**

Each partition will be written to a separate file.

**Examples**

```
>>> df = dd.read_csv(...)
>>> to_parquet('/path/to/output/', df, compression='snappy')
```

**to\_string** (*max\_rows=5*)

Render a string representation of the Series

**Parameters** **buf** : StringIO-like, optional

buffer to write to

**na\_rep** : string, optional

string representation of NAN to use, default 'NaN'

**float\_format** : one-parameter function, optional

formatter function to apply to columns' elements if they are floats default None

**header: boolean, default True**

Add the Series header (index name)

**index** : bool, optional

Add index (row) labels, default True

**length** : boolean, default False

Add the Series length

**dtype** : boolean, default False

Add the Series dtype

**name** : boolean, default False

Add the Series name if not None

**max\_rows** : int, optional

Maximum number of rows to show before truncating. If None, show all.

**Returns** **formatted** : string (if not buffer passed)

## Notes

Dask doesn't support the following argument(s).

- buf
- na\_rep
- float\_format
- header
- index
- length
- dtype
- name

**to\_timestamp** (*freq=None, how='start', axis=0*)

Cast to DatetimeIndex of timestamps, at *beginning* of period

**Parameters** **freq** : string, default frequency of PeriodIndex

Desired frequency

**how** : {'s', 'e', 'start', 'end'}

Convention for converting period to timestamp; start of period vs. end

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

**copy** : boolean, default True

If false then underlying input data is not copied

**Returns** **df** : DataFrame with DatetimeIndex

## Notes

Dask doesn't support the following argument(s).

- copy

**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** (*split\_every=None, split\_out=1*)

Return Series of unique values in the object. Includes NA values.

**Returns uniques** : Series

**value\_counts** (*split\_every=None, split\_out=1*)

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 support the following argument(s).

- normalize
- sort
- ascending
- bins
- dropna

## values

Return a `dask.array` of the values of this dataframe

Warning: This creates a `dask.array` without precise shape information. Operations that depend on shape information, like slicing or reshaping, will not work.

**var** (*axis=None, skipna=True, ddof=1, split\_every=False*)

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 columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns** `var` : Series or DataFrame (if level specified)

## Notes

Dask doesn't support the 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.display.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-like, or callable

Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. 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

Entries where *cond* is False are replaced with corresponding value from *other*. 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

**errors** : str, {'raise', 'ignore'}, default 'raise'

- *raise* : allow exceptions to be raised
- *ignore* : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**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)

Deprecated since version 0.21.0.

**Returns** *wh* : same type as caller

**See also:**

*DataFrame.mask()*

## Notes

The `where` method is an application of the if-then idiom. For each element in the calling `DataFrame`, if `cond` is `True` the element is used; otherwise the corresponding element from the `DataFrame` `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in [indexing](#).

## Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
```

```
>>> s.mask(s > 0)
0     0.0
1     NaN
2     NaN
3     NaN
4     NaN
```

```
>>> s.where(s > 1, 10)
0     10.0
1     10.0
2     2.0
3     3.0
4     4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A  B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
   A  B
0  True  True
1  True  True
2  True  True
```

```
3 True True
4 True True
```

## DataFrameGroupBy

**class** `dask.dataframe.groupby.DataFrameGroupBy` (*df, by=None, slice=None*)

**agg** (*arg, split\_every=None, split\_out=1*)

Aggregate using callable, string, dict, or list of string/callables

**Parameters** `func` : callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

**Returns** `aggregated` : DataFrame

**See also:**

`pandas.DataFrame.groupby.apply`, `pandas.DataFrame.groupby.transform`,  
`pandas.DataFrame.aggregate`

## Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., `np.mean(arr_2d, axis=0)`) as opposed to mimicking the default Numpy behavior (e.g., `np.mean(arr_2d)`).

`agg` is an alias for `aggregate`. Use the alias.

## Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 2],
...                    'B': [1, 2, 3, 4],
...                    'C': np.random.randn(4)})
```

```
>>> df
   A  B      C
0  1  1  0.362838
1  1  2  0.227877
2  2  3  1.267767
3  2  4 -0.562860
```

The aggregation is for each column.

```
>>> df.groupby('A').agg('min')
      B          C
A
1  1  0.227877
2  3 -0.562860
```

#### Multiple aggregations

```
>>> df.groupby('A').agg(['min', 'max'])
      B          C
      min max      min      max
A
1  1  2  0.227877  0.362838
2  3  4 -0.562860  1.267767
```

#### Select a column for aggregation

```
>>> df.groupby('A').B.agg(['min', 'max'])
      min  max
A
1  1  2
2  3  4
```

#### Different aggregations per column

```
>>> df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})
      B          C
      min max      sum
A
1  1  2  0.590716
2  3  4  0.704907
```

**aggregate** (*arg*, *split\_every=None*, *split\_out=1*)

Aggregate using callable, string, dict, or list of string/callables

**Parameters** *func* : callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

**Returns** *aggregated* : DataFrame

#### See also:

`pandas.DataFrame.groupby.apply`, `pandas.DataFrame.groupby.transform`,  
`pandas.DataFrame.aggregate`

## Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., `np.mean(arr_2d, axis=0)`) as opposed to mimicking the default Numpy behavior (e.g., `np.mean(arr_2d)`).

`agg` is an alias for `aggregate`. Use the alias.

## Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 2],
...                    'B': [1, 2, 3, 4],
...                    'C': np.random.randn(4)})
```

```
>>> df
   A  B      C
0  1  1  0.362838
1  1  2  0.227877
2  2  3  1.267767
3  2  4 -0.562860
```

The aggregation is for each column.

```
>>> df.groupby('A').agg('min')
   B      C
A
1  1  0.227877
2  3 -0.562860
```

### Multiple aggregations

```
>>> df.groupby('A').agg(['min', 'max'])
   B      C
   min max      min      max
A
1  1  2  0.227877  0.362838
2  3  4 -0.562860  1.267767
```

Select a column for aggregation

```
>>> df.groupby('A').B.agg(['min', 'max'])
   min  max
A
1     1     2
2     3     4
```

Different aggregations per column

```
>>> df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})
   B      C
   min max      sum
A
1  1  2  0.590716
2  3  4  0.704907
```

**apply** (*func*, *meta*='\_\_no\_default\_\_')

Parallel version of pandas GroupBy.apply

This mimics the pandas version except for the following:

1. The user should provide output metadata.
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

**meta** : pd.DataFrame, pd.Series, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**Returns applied** : Series or DataFrame depending on columns keyword

**count** (*split\_every*=None, *split\_out*=1)

Compute count of group, excluding missing values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**cumcount** (*axis*=None)

Number each item in each group from 0 to the length of that group - 1.

Essentially this is equivalent to

```
>>> self.apply(lambda x: Series(np.arange(len(x)), x.index))
```

**Parameters ascending** : bool, default True

If False, number in reverse, from length of group - 1 to 0.

**See also:**

**ngroup** Number the groups themselves.

**Notes**

Dask doesn't support the following argument(s).

- ascending

## Examples

```
>>> df = pd.DataFrame([[ 'a' ], [ 'a' ], [ 'a' ], [ 'b' ], [ 'b' ], [ 'a' ]],
...                    columns=[ 'A' ])
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby( 'A' ).cumcount()
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64
>>> df.groupby( 'A' ).cumcount( ascending=False )
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64
```

### **cumprod** (*axis=0*)

Cumulative product for each group

#### **See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

### **cumsum** (*axis=0*)

Cumulative sum for each group

#### **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 support the following argument(s).

- name
- obj

**max** (*split\_every=None, split\_out=1*)  
Compute max of group values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**mean** (*split\_every=None, split\_out=1*)  
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** (*split\_every=None, split\_out=1*)  
Compute min of group values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**size** (*split\_every=None, split\_out=1*)  
Compute group sizes

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**std** (*ddof=1, split\_every=None, split\_out=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** (*split\_every=None, split\_out=1*)  
Compute sum of group values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**var** (*ddof=1, split\_every=None, split\_out=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*, *by=None*, *slice=None*)

**agg** (*arg*, *split\_every=None*, *split\_out=1*)

Aggregate using callable, string, dict, or list of string/callables

**Parameters** **func** : callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

**Returns** **aggregated** : Series

**See also:**

`pandas.Series.groupby.apply`, `pandas.Series.groupby.transform`, `pandas.Series.aggregate`

## Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., `np.mean(arr_2d, axis=0)`) as opposed to mimicking the default Numpy behavior (e.g., `np.mean(arr_2d)`).

`agg` is an alias for `aggregate`. Use the alias.

## Examples

```
>>> s = Series([1, 2, 3, 4])
```

```
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.groupby([1, 1, 2, 2]).min()
1    1
2    3
dtype: int64
```

```
>>> s.groupby([1, 1, 2, 2]).agg('min')
1    1
2    3
dtype: int64
```

```
>>> s.groupby([1, 1, 2, 2]).agg(['min', 'max'])
      min  max
1      1    2
2      3    4
```

**aggregate** (*arg*, *split\_every=None*, *split\_out=1*)

Aggregate using callable, string, dict, or list of string/callables

**Parameters** **func** : callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

**Returns** **aggregated** : Series

**See also:**

`pandas.Series.groupby.apply`, `pandas.Series.groupby.transform`, `pandas.Series.aggregate`

## Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., `np.mean(arr_2d, axis=0)`) as opposed to mimicking the default Numpy behavior (e.g., `np.mean(arr_2d)`).

`agg` is an alias for `aggregate`. Use the alias.

## Examples

```
>>> s = Series([1, 2, 3, 4])
```

```
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.groupby([1, 1, 2, 2]).min()
1    1
2    3
dtype: int64
```

```
>>> s.groupby([1, 1, 2, 2]).agg('min')
1    1
2    3
dtype: int64
```

```
>>> s.groupby([1, 1, 2, 2]).agg(['min', 'max'])
   min  max
1     1    2
2     3    4
```

**apply** (*func*, *meta*='\_\_no\_default\_\_')

Parallel version of pandas GroupBy.apply

This mimics the pandas version except for the following:

1. The user should provide output metadata.
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

**meta** : pd.DataFrame, pd.Series, dict, iterable, tuple, optional

An empty pd.DataFrame or pd.Series that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a DataFrame, a dict of {name: dtype} or iterable of (name, dtype) can be provided. Instead of a series, a tuple of (name, dtype) can be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing meta is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**Returns** **applied** : Series or DataFrame depending on columns keyword

**count** (*split\_every*=None, *split\_out*=1)

Compute count of group, excluding missing values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**cumcount** (*axis*=None)

Number each item in each group from 0 to the length of that group - 1.

Essentially this is equivalent to

```
>>> self.apply(lambda x: Series(np.arange(len(x)), x.index))
```

**Parameters** **ascending** : bool, default True

If False, number in reverse, from length of group - 1 to 0.

**See also:**

**nngroup** Number the groups themselves.

**Notes**

Dask doesn't support the following argument(s).

- ascending

**Examples**

```
>>> df = pd.DataFrame(['a'], ['a'], ['a'], ['b'], ['b'], ['a']],
...                    columns=['A'])
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby('A').cumcount()
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64
>>> df.groupby('A').cumcount(ascending=False)
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64
```

**cumprod** (*axis=0*)

Cumulative product for each group

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**cumsum** (*axis=0*)

Cumulative sum for each group

**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 support the following argument(s).

- name
- obj

**max** (*split\_every=None, split\_out=1*)  
Compute max of group values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**mean** (*split\_every=None, split\_out=1*)  
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** (*split\_every=None, split\_out=1*)  
Compute min of group values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**size** (*split\_every=None, split\_out=1*)  
Compute group sizes

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**std** (*ddof=1, split\_every=None, split\_out=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** (*split\_every=None, split\_out=1*)  
Compute sum of group values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

**var** (*ddof=1, split\_every=None, split\_out=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`

## Storage and Conversion

`dask.dataframe.read_csv` (*urlpath, blocksize=64000000, collection=True, lineterminator=None, compression=None, sample=256000, enforce=False, assume\_missing=False, storage\_options=None, \*\*kwargs*)

Read CSV files into a `Dask.DataFrame`

This parallelizes the `pandas.read_csv` function in the following ways:

- It supports loading many files at once using globstrings:

```
>>> df = dd.read_csv('myfiles.*.csv')
```

- In some cases it can break up large files:

```
>>> df = dd.read_csv('largefile.csv', blocksize=25e6) # 25MB chunks
```

- It can read CSV files from external resources (e.g. S3, HDFS) by 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 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.

**Parameters** **urlpath** : string

Absolute or relative filepath, URL (may include protocols like `s3://`), or globstring for CSV files.

**blocksize** : int or None, optional

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, optional

Return a `dask.dataframe` if True or list of `dask.delayed` objects if False

**sample** : int, optional

Number of bytes to use when determining dtypes

**assume\_missing** : bool, optional

If True, all integer columns that aren't specified in `dtype` are assumed to contain missing values, and are converted to floats. Default is False.

**storage\_options** : dict, optional

Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc.

**\*\*kwargs**

Extra keyword arguments to forward to `pandas.read_csv`.

## Notes

Dask dataframe tries to infer the `dtype` of each column by reading a sample from the start of the file (or of the first file if it's a glob). Usually this works fine, but if the `dtype` is different later in the file (or in other files) this can cause issues. For example, if all the rows in the sample had integer dtypes, but later on there was a `NaN`, then this would error at compute time. To fix this, you have a few options:

- Provide explicit dtypes for the offending columns using the `dtype` keyword. This is the recommended solution.
- Use the `assume_missing` keyword to assume that all columns inferred as integers contain missing values, and convert them to floats.
- Increase the size of the sample using the `sample` keyword.

It should also be noted that this function may fail if a CSV file includes quoted strings that contain the line terminator. To get around this you can specify `blocksize=None` to not split files into multiple partitions, at the cost of reduced parallelism.

```
dask.dataframe.read_table(urlpath, blocksize=64000000, collection=True, lineterminator=None, compression=None, sample=256000, enforce=False, assume_missing=False, storage_options=None, **kwargs)
```

Read delimited files into a `Dask.DataFrame`

This parallelizes the `pandas.read_table` function in the following ways:

- It supports loading many files at once using globstrings:

```
>>> df = dd.read_table('myfiles.*.csv')
```

- In some cases it can break up large files:

```
>>> df = dd.read_table('largefile.csv', blocksize=25e6) # 25MB chunks
```

- It can read CSV files from external resources (e.g. S3, HDFS) by providing a URL:

```
>>> df = dd.read_table('s3://bucket/myfiles.*.csv')
>>> df = dd.read_table('hdfs:///myfiles.*.csv')
>>> df = dd.read_table('hdfs://namenode.example.com/myfiles.*.csv')
```

Internally `dd.read_table` uses `pandas.read_table` and supports many of the same keyword arguments with the same performance guarantees. See the docstring for `pandas.read_table` for more information on available keyword arguments.

**Parameters urlpath** : string

Absolute or relative filepath, URL (may include protocols like `s3://`), or globstring for delimited files.

**blocksize** : int or None, optional

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, optional

Return a `dask.dataframe` if `True` or list of `dask.delayed` objects if `False`

**sample** : int, optional

Number of bytes to use when determining dtypes

**assume\_missing** : bool, optional

If `True`, all integer columns that aren't specified in `dtype` are assumed to contain missing values, and are converted to floats. Default is `False`.

**storage\_options** : dict, optional

Extra options that make sense for a particular storage connection, e.g. `host`, `port`, `username`, `password`, etc.

**\*\*kwargs**

Extra keyword arguments to forward to `pandas.read_table`.

## Notes

Dask dataframe tries to infer the `dtype` of each column by reading a sample from the start of the file (or of the first file if it's a glob). Usually this works fine, but if the `dtype` is different later in the file (or in other files) this can cause issues. For example, if all the rows in the sample had integer dtypes, but later on there was a `NaN`, then this would error at compute time. To fix this, you have a few options:

- Provide explicit dtypes for the offending columns using the `dtype` keyword. This is the recommended solution.
- Use the `assume_missing` keyword to assume that all columns inferred as integers contain missing values, and convert them to floats.
- Increase the size of the sample using the `sample` keyword.

It should also be noted that this function may fail if a delimited file includes quoted strings that contain the line terminator. To get around this you can specify `blocksize=None` to not split files into multiple partitions, at the cost of reduced parallelism.

```
dask.dataframe.read_parquet(path, columns=None, filters=None, categories=None, index=None,
                           storage_options=None, engine='auto')
```

Read ParquetFile into a Dask DataFrame

This reads a directory of Parquet data into a `Dask.dataframe`, one file per partition. It selects the index among the sorted columns if any exist.

**Parameters path** : string

Source directory for data. May be a glob string. Prepend with protocol like `s3://` or `hdfs://` for remote data.

**columns: list or None**

List of column names to load

**filters: list**

List of filters to apply, like `[('x', '>', 0), ...]`. This implements row-group (partition) -level filtering only, i.e., to prevent the loading of some chunks of the data, and only if relevant statistics have been included in the metadata.

**index:** string or None (default) or False

Name of index column to use if that column is sorted; False to force dask to not use any column as the index

**categories:** list, dict or None

For any fields listed here, if the parquet encoding is Dictionary, the column will be created with dtype category. Use only if it is guaranteed that the column is encoded as dictionary in all row-groups. If a list, assumes up to  $2^{16}-1$  labels; if a dict, specify the number of labels expected; if None, will load categories automatically for data written by dask/fastparquet, not otherwise.

**storage\_options :** dict

Key/value pairs to be passed on to the file-system backend, if any.

**engine :** {'auto', 'fastparquet', 'pyarrow'}, default 'auto'

Parquet reader library to use. If only one library is installed, it will use that one; if both, it will use 'fastparquet'

**See also:**

`to_parquet`

## Examples

```
>>> df = read_parquet('s3://bucket/my-parquet-data')
```

`dask.dataframe.read_hdf` (*pattern*, *key*, *start=0*, *stop=None*, *columns=None*, *chunksize=1000000*, *sorted\_index=False*, *lock=True*, *mode='a'*)

Read HDF files into a Dask DataFrame

Read hdf files into a dask dataframe. This function is like `pandas.read_hdf`, except it can read from a single large file, or from multiple files, or from multiple keys from the same file.

**Parameters** *pattern* : string, list

File pattern (string), buffer to read from, or list of file paths. Can contain wildcards.

**key** : group identifier in the store. Can contain wildcards

**start** : optional, integer (defaults to 0), row number to start at

**stop** : optional, integer (defaults to None, the last row), row number to stop at

**columns** : list of columns, optional

A list of columns that if not None, will limit the return columns (default is None)

**chunksize** : positive integer, optional

Maximal number of rows per partition (default is 1000000).

**sorted\_index** : boolean, optional

Option to specify whether or not the input hdf files have a sorted index (default is False).

**lock** : boolean, optional

Option to use a lock to prevent concurrency issues (default is True).

**mode** : {'a', 'r', 'r+'}, default 'a'. Mode to use when opening file(s).

'r' Read-only; no data can be modified.

'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.

**Returns** dask.DataFrame

## Examples

Load single file

```
>>> dd.read_hdf('myfile.1.hdf5', '/x')
```

Load multiple files

```
>>> dd.read_hdf('myfile.*.hdf5', '/x')
```

```
>>> dd.read_hdf(['myfile.1.hdf5', 'myfile.2.hdf5'], '/x')
```

Load multiple datasets

```
>>> dd.read_hdf('myfile.1.hdf5', '/*')
```

```
dask.dataframe.read_sql_table(table, uri, index_col, divisions=None, npartitions=None,
                              limits=None, columns=None, bytes_per_chunk=268435456,
                              **kwargs)
```

Create dataframe from an SQL table.

If neither divisions or npartitions is given, the memory footprint of the first five rows will be determined, and partitions of size ~256MB will be used.

**Parameters** **table** : string or sqlalchemy expression

Select columns from here.

**uri** : string

Full sqlalchemy URI for the database connection

**index\_col** : string

Column which becomes the index, and defines the partitioning. Should be a indexed column in the SQL server, and numerical. Could be a function to return a value, e.g., `sql.func.abs(sql.column('value')).label('abs(value)')`. Labeling columns created by functions or arithmetic operations is required.

**divisions**: sequence

Values of the index column to split the table by.

**npartitions** : int

Number of partitions, if divisions is not given. Will split the values of the index column linearly between limits, if given, or the column max/min.

**limits**: 2-tuple or None

Manually give upper and lower range of values for use with npartitions; if None, first fetches max/min from the DB. Upper limit, if given, is inclusive.

**columns** : list of strings or None

Which columns to select; if None, gets all; can include sqlalchemy functions, e.g., `sql.func.abs(sql.column('value')).label('abs(value)')`. Labeling columns created by functions or arithmetic operations is recommended.

**bytes\_per\_chunk**: int

If both divisions and npartitions is None, this is the target size of each partition, in bytes

**kwargs** : dict

Additional parameters to pass to `pd.read_sql()`

**Returns** dask.dataframe

## Examples

```
>>> df = dd.read_sql('accounts', 'sqlite:///path/to/bank.db',
...                 npartitions=10, index_col='id')
```

`dask.dataframe.from_array(x, chunksize=50000, columns=None)`

Read any slicable array into a Dask Dataframe

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 DataFrame from a Pandas DataFrame

This splits an in-memory Pandas dataframe into several parts and constructs a `dask.dataframe` from those parts on which `Dask.dataframe` can operate in parallel.

Note that, despite parallelism, `Dask.dataframe` may not always be faster than Pandas. We recommend that you stay with Pandas for as long as possible before switching to `Dask.dataframe`.

**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. Note that depending on the size and index of the dataframe, the output may have fewer partitions than requested.

**chunksize** : int, optional

The number of rows per index partition to use.

**sort**: bool

Sort input first to obtain cleanly divided partitions or don't sort and don't get cleanly divided partitions

**name**: string, optional

An optional keyname for the dataframe. Defaults to hashing the input

**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

`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', freq='D'),
 timestamp('2010-01-03 00:00:00', freq='D'),
 timestamp('2010-01-05 00:00:00', freq='D'),
 timestamp('2010-01-06 00:00:00', freq='D'))
>>> ddf = from_pandas(df.a, npartitions=3) # Works with Series too!
>>> ddf.divisions
(timestamp('2010-01-01 00:00:00', freq='D'),
 timestamp('2010-01-03 00:00:00', freq='D'),
 timestamp('2010-01-05 00:00:00', freq='D'),
 timestamp('2010-01-06 00:00:00', freq='D'))
```

`dask.dataframe.from_bcolz(x, chunksize=None, categorize=True, index=None, lock=<unlocked  
_thread.lock object>, **kwargs)`

Read BColz CTable into a Dask Dataframe

BColz is a fast on-disk compressed column store with careful attention given to compression. <https://bcolz.readthedocs.io/en/latest/>

**Parameters** `x` : bcolz.ctable

**chunksize** : int, optional

The size(rows) of blocks to pull out from ctable.

**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.from_dask_array(x, columns=None)`

Create a Dask DataFrame from a Dask Array.

Converts a 2d array into a DataFrame and a 1d array into a Series.

**Parameters** `x`: da.Array

**columns**: list or string

list of column names if DataFrame, single string if Series

**See also:**

`dask.bag.to_dataframe` from dask.bag

`dask.dataframe._Frame.values` Reverse conversion

`dask.dataframe._Frame.to_records` Reverse conversion

## Examples

```
>>> import dask.array as da
>>> import dask.dataframe as dd
>>> x = da.ones((4, 2), chunks=(2, 2))
>>> df = dd.io.from_dask_array(x, columns=['a', 'b'])
>>> df.compute()
   a  b
0  1.0 1.0
1  1.0 1.0
2  1.0 1.0
3  1.0 1.0
```

`dask.dataframe.from_delayed(dfs, meta=None, divisions=None, prefix='from-delayed')`

Create Dask DataFrame from many Dask Delayed objects

**Parameters** `dfs` : list of Delayed

An iterable of `dask.delayed.Delayed` objects, such as come from `dask.delayed`. These comprise the individual partitions of the resulting dataframe.

**meta** : pd.DataFrame, pd.Series, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in dask dataframe to work. For ease of use, some alternative inputs are also available. Instead of a DataFrame, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can

be used. If not provided, dask will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

**divisions** : tuple, str, optional

Partition boundaries along the index. For tuple, see <http://dask.pydata.org/en/latest/dataframe-design.html#partitions> For string 'sorted' will compute the delayed values to find index values. Assumes that the indexes are mutually sorted. If None, then won't use index information

**prefix** : str, optional

Prefix to prepend to the keys.

`dask.dataframe.to_delayed(df)`  
 Create Dask Delayed objects from a Dask Dataframe  
 Returns a list of delayed values, one value per partition.

### Examples

```
>>> partitions = df.to_delayed()
```

`dask.dataframe.to_records(df)`  
 Create Dask Array from a Dask Dataframe  
 Warning: This creates a `dask.array` without precise shape information. Operations that depend on shape information, like slicing or reshaping, will not work.

**See also:**

`dask.dataframe._Frame.values`, `dask.dataframe.from_dask_array`

### Examples

```
>>> df.to_records()
dask.array<shape=(nan,), dtype=(numpy.record, [('ind', '<f8'), ('x', 'O'), ('y', '
↳<i8')])>, chunksize=(nan,>
```

`dask.dataframe.to_csv(df, filename, name_function=None, compression=None, compute=True, get=None, storage_options=None, **kwargs)`  
 Store Dask DataFrame to CSV files

One filename per partition will be created. You can specify the filenames in a variety of ways.

Use a globstring:

```
>>> df.to_csv('/path/to/data/export-*.csv')
```

The `*` will be replaced by the increasing sequence 0, 1, 2, ...

```
/path/to/data/export-0.csv
/path/to/data/export-1.csv
```

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'
```

```
>>> df.to_csv('/path/to/data/export-*.csv', name_function=name)
```

```
/path/to/data/export-2015-01-01.csv
/path/to/data/export-2015-01-02.csv
...
```

You can also provide an explicit list of paths:

```
>>> paths = ['/path/to/data/alice.csv', '/path/to/data/bob.csv', ...]
>>> df.to_csv(paths)
```

**Parameters filename** : string

Path glob indicating the naming scheme for the output files

**name\_function** : callable, default None

Function accepting an integer (partition index) and producing a string to replace the asterisk in the given filename globstring. Should preserve the lexicographic order of partitions

**compression** : string or None

String like 'gzip' or 'xz'. Must support efficient random access. Filenames with extensions corresponding to known compression algorithms (gz, bz2) will be compressed accordingly automatically

**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

**storage\_options**: dict

Parameters passed on to the backend filesystem class.

**Returns**

—

The names of the file written if they were computed right away

**If not, the delayed tasks associated to the writing of the files**`dask.dataframe.to_bag(df, index=False)`

Create Dask Bag from a Dask DataFrame

**Parameters** `index` : bool, optionalIf True, the elements are tuples of (`index`, `value`), otherwise they're just the `value`. Default is False.**Examples**

```
>>> bag = df.to_bag()
```

`dask.dataframe.to_hdf(df, path, key, mode='a', append=False, get=None, name_function=None, compute=True, lock=None, dask_kwargs={}, **kwargs)`

Store Dask Dataframe to Hierarchical Data Format (HDF) files

This is a parallel version of the Pandas function of the same name. Please see the Pandas docstring for more detailed information about shared keyword arguments.

This function differs from the Pandas version by saving the many partitions of a Dask DataFrame in parallel, either to many files, or to many datasets within the same file. You may specify this parallelism with an asterisk `*` within the filename or datapath, and an optional `name_function`. The asterisk will be replaced with an increasing sequence of integers starting from 0 or with the result of calling `name_function` on each of those integers.

This function only supports the Pandas 'table' format, not the more specialized 'fixed' format.

**Parameters** `path`: stringPath to a target filename. May contain a `*` to denote many filenames**key**: stringDatapath within the files. May contain a `*` to denote many locations**name\_function**: functionA function to convert the `*` in the above options to a string. Should take in a number from 0 to the number of partitions and return a string. (see examples below)**compute**: boolWhether or not to execute immediately. If False then this returns a `dask.Delayed` value.**lock**: Lock, optionalLock to use to prevent concurrency issues. By default a `threading.Lock`, `multiprocessing.Lock` or `SerializableLock` will be used depending on your scheduler if a lock is required. See `dask.utils.get_scheduler_lock` for more information about lock selection.**\*\*other**:See `pandas.to_hdf` for more information**Returns** None: if `compute == True`delayed value: if `compute == False`

**See also:**`read_hdf, to_parquet`**Examples**

Save Data to a single file

```
>>> df.to_hdf('output.hdf', '/data')
```

Save data to multiple datapaths within the same file:

```
>>> df.to_hdf('output.hdf', '/data-*')
```

Save data to multiple files:

```
>>> df.to_hdf('output-*.hdf', '/data')
```

Save data to multiple files, using the multiprocessing scheduler:

```
>>> df.to_hdf('output-*.hdf', '/data', get=dask.multiprocessing.get)
```

Specify custom naming scheme. This writes files as '2000-01-01.hdf', '2000-01-02.hdf', '2000-01-03.hdf', etc..

```
>>> from datetime import date, timedelta
>>> base = date(year=2000, month=1, day=1)
>>> def name_function(i):
...     ''' Convert integer 0 to n to a string '''
...     return base + timedelta(days=i)
```

```
>>> df.to_hdf('*.hdf', '/data', name_function=name_function)
```

`dask.dataframe.to_parquet` (*df*, *path*, *engine*='auto', *compression*='default', *write\_index*=None, *append*=False, *ignore\_divisions*=False, *partition\_on*=None, *storage\_options*=None, *compute*=True, *\*\*kwargs*)

Store Dask.dataframe to Parquet files

**Parameters** *df*: `dask.dataframe.DataFrame`**path**: stringDestination directory for data. Prepend with protocol like `s3://` or `hdfs://` for remote data.**engine**: {'auto', 'fastparquet', 'pyarrow'}, default 'auto'

Parquet library to use. If only one library is installed, it will use that one; if both, it will use 'fastparquet'.

**compression**: string or dict, optional

Either a string like "snappy" or a dictionary mapping column names to compressors like {"name": "gzip", "values": "snappy"}. The default is "default", which uses the default compression for whichever engine is selected.

**write\_index**: boolean, optionalWhether or not to write the index. Defaults to True *if* divisions are known.

**append** : bool, optional

If False (default), construct data-set from scratch. If True, add new row-group(s) to an existing data-set. In the latter case, the data-set must exist, and the schema must match the input data.

**ignore\_divisions** : bool, optional

If False (default) raises error when previous divisions overlap with the new appended divisions. Ignored if `append=False`.

**partition\_on** : list, optional

Construct directory-based partitioning by splitting on these fields' values. Each dask partition will result in one or more datafiles, there will be no global groupby.

**storage\_options** : dict, optional

Key/value pairs to be passed on to the file-system backend, if any.

**compute** : bool, optional

If True (default) then the result is computed immediately. If False then a `dask.delayed` object is returned for future computation.

**\*\*kwargs**

Extra options to be passed on to the specific backend.

See also:

[`read\_parquet`](#) Read parquet data to `dask.dataframe`

## Notes

Each partition will be written to a separate file.

## Examples

```
>>> df = dd.read_csv(...)
>>> to_parquet('/path/to/output/', df, compression='snappy')
```

## Rolling

`dask.dataframe.rolling.map_overlap` (*func, df, before, after, \*args, \*\*kwargs*)

Apply a function to each partition, sharing rows with adjacent partitions.

**Parameters** **func** : function

Function applied to each partition.

**df** : `dd.DataFrame`, `dd.Series`

**before** : int or `timedelta`

The rows to prepend to partition *i* from the end of partition *i* - 1.

**after** : int or `timedelta`

The rows to append to partition *i* from the beginning of partition *i* + 1.

**args, kwargs :**

Arguments and keywords to pass to the function. The partition will be the first argument, and these will be passed *after*.

**See also:**

`dd.DataFrame.map_overlap`

**Other functions**

`dask.dataframe.compute(*args, **kwargs)`

Compute several dask collections at once.

**Parameters args :** object

Any number of objects. If it is a dask object, it's computed and the result is returned. By default, python builtin collections are also traversed to look for dask objects (for more information see the `traverse` keyword). Non-dask arguments are passed through unchanged.

**traverse :** bool, optional

By default dask traverses builtin python collections looking for dask objects passed to `compute`. For large collections this can be expensive. If none of the arguments contain any dask objects, set `traverse=False` to avoid doing this traversal.

**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)
```

By default, dask objects inside python collections will also be computed:

```
>>> compute({'a': a, 'b': b, 'c': 1})
({'a': 45, 'b': 4.5, 'c': 1},)
```

`dask.dataframe.map_partitions(func, *args, **kwargs)`

Apply Python function on each DataFrame partition.

**Parameters func :** function

Function applied to each partition.

**args, kwargs :**

Arguments and keywords to pass to the function. At least one of the args should be a `Dask.dataframe`.

**meta** : `pd.DataFrame`, `pd.Series`, dict, iterable, tuple, optional

An empty `pd.DataFrame` or `pd.Series` that matches the dtypes and column names of the output. This metadata is necessary for many algorithms in `dask.dataframe` to work. For ease of use, some alternative inputs are also available. Instead of a `DataFrame`, a dict of `{name: dtype}` or iterable of `(name, dtype)` can be provided. Instead of a series, a tuple of `(name, dtype)` can be used. If not provided, `dask` will try to infer the metadata. This may lead to unexpected results, so providing `meta` is recommended. For more information, see `dask.dataframe.utils.make_meta`.

`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.

**Notes**

This differs in from `pd.concat` in the when concatenating Categoricals with different categories. Pandas currently coerces those to objects before concatenating. Coercing to objects is very expensive for large arrays, so `dask` preserves the Categoricals by taking the union of the categories.

**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)>
```

Different categoricals are unioned

```
>> dd.concat([ # doctest: +SKIP ... dd.from_pandas(pd.Series(['a', 'b'], dtype='category'), 1), ...
dd.from_pandas(pd.Series(['a', 'c'], dtype='category'), 1), ... ], interleave_partitions=True).dtype CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)
```

`dask.dataframe.multi.merge` (*left*, *right*, *how*='inner', *on*=None, *left\_on*=None, *right\_on*=None, *left\_index*=False, *right\_index*=False, *sort*=False, *suffixes*=('\_x', '\_y'), *copy*=True, *indicator*=False, *validate*=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** *left* : DataFrame

**right** : DataFrame

**how** : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

**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. If False, the order of the join keys depends on the join type (how keyword)

**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.

**validate** : string, default None

If specified, checks if merge is of specified type.

- “one\_to\_one” or “1:1”: check if merge keys are unique in both left and right datasets.
- “one\_to\_many” or “1:m”: check if merge keys are unique in left dataset.
- “many\_to\_one” or “m:1”: check if merge keys are unique in right dataset.
- “many\_to\_many” or “m:m”: allowed, but does not result in checks.

New in version 0.21.0.

**Returns** **merged** : DataFrame

The output type will be the same as ‘left’, if it is a subclass of DataFrame.

**See also:**

merge\_ordered, merge\_asof

**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
```

## 4.8.4 Dask DataFrame Performance Tips

### Use Pandas

For data that fits into RAM, Pandas can often be faster and easier to use than Dask.dataframe. While “Big Data” tools can be exciting, they are almost always worse than normal data tools while those remain appropriate.

### Pandas Performance Tips Apply to Dask.dataframe

Normal Pandas performance tips, like avoiding apply, using vectorized operations, using categoricals, etc. all apply equally to Dask.dataframe. See [Modern Pandas](#) by Tom Augspurger is a good read here.

### Use the Index

Dask.dataframe can be optionally sorted along a single index column. Some operations against this column can be very fast. For example if your dataset is sorted by time you can quickly select data for a particular day, perform time series joins, etc. You can check if your data is sorted by looking at the `df.known_divisions` attribute. You can set an index column using the `.set_index(columnname)` method. This operation is expensive though, so use it sparingly (see below).

```
df = df.set_index('timestamp') # set the index to make some operations fast

df.loc['2001-01-05':'2001-01-12'] # this is very fast if you have an index
df.merge(df2, left_index=True, right_index=True) # this is also very fast
```

### Avoid Shuffles

Setting an index is an important (see above) but expensive operation. You should do it infrequently and you should persist afterwards (see below).

Some operations like `set_index` and `merge/join` are harder to do in a parallel or distributed setting than they are in-memory on a single machine. In particular *shuffling operations* that rearrange data become much more communication intensive. For example if your data is arranged by customer ID but now you want to arrange it by time all of your partitions will have to talk to each other to exchange shards of data. This can be an intense process, particularly on a cluster.

So definitely set the index, but try do so infrequently. After you set the index then you may want to `persist` your data if you are on a cluster.

```
df = df.set_index('column-name') # do this infrequently
```

Additionally, `set_index` has a few options that can accelerate it in some situations. For example if you know that your dataset is sorted or you already know the values by which it is divided you can provide these to accelerate the `set_index` operation. See the `set_index` docstring for more information.

```
df2 = df.set_index(d.timestamp, sorted=True)
```

## Persist Intelligently

*This section is only relevant to users on distributed systems.*

Often dataframe workloads look like the following:

1. Load data from files
2. Filter data to a particular subset
3. Shuffle data to set an intelligent index
4. Several complex queries on top of this indexed data

It is often ideal to load, filter, and shuffle data once and keep this result in memory. Afterwards each of the several complex queries can be based off of this in-memory data rather than have to repeat the full load-filter-shuffle process each time. To do this, use the `client.persist` method.

```
df = dd.read_csv('s3://bucket/path/to/*.csv')
df = df[df.balance < 0]
df = client.persist(df)

df = df.set_index('timestamp')
df = client.persist(df)

>>> df.customer_id.nunique().compute()
18452844

>>> df.groupby(df.city).size().compute()
...
```

Persist is important because `Dask.dataframe` is *lazy by default*. Persist is a way of telling the cluster that it should start computing on the computations that you have defined so far and that it should try to keep those results in memory. You will get back a new dataframe that is semantically equivalent to your old dataframe, but now points to running data. Your old dataframe still points to lazy computations

```
# Don't do this
client.persist(df) # Persist doesn't change the input in-place

# Do this instead
df = client.persist(df) # Replace your old lazy dataframe
```

## Repartition to Reduce Overhead

Your `Dask.dataframe` is split up into many Pandas dataframes. We sometimes call these “partitions”. Often the number of partitions is decided for you; for example it might be the number of CSV files from which you are reading. However over time as you reduce or increase the size of your pandas dataframes by filtering or joining it may be wise to reconsider how many partitions you need. There is a cost to having too many or having too few.

Partitions should fit comfortably in memory (smaller than a gigabyte) but also not be too numerous. Every operation on every partition takes the central scheduler a few hundred microseconds to process. If you have a few thousand tasks this is barely noticeable, but it is nice to reduce the number if possible.

A common situation is that you load lots of data into reasonably sized partitions (dask’s defaults make decent choices) but then you filter down your dataset to only a small fraction of the original. At this point it is wise to regroup your many small partitions into a few larger ones. You can do this with the `repartition` method:

```
df = dd.read_csv('s3://bucket/path/to/*.csv')
df = df[df.name == 'Alice'] # only 1/100th of the data
df = df.repartition(npartitions=df.npartitions // 100)

df = client.persist(df) # if on a distributed system
```

This helps to reduce overhead and increase the effectiveness of vectorized Pandas operations. You should aim for partitions that have around 100MB of data each.

Additionally, reducing partitions is very helpful just before shuffling, which creates  $n \log(n)$  tasks relative to the number of partitions. Dataframes with less than 100 partitions are much easier to shuffle than dataframes with tens of thousands.

## Joins

Joining two dataframes can be either very expensive or very cheap depending on the situation. It is cheap in the following cases:

1. Joining a `Dask.dataframe` with a Pandas dataframe
2. Joining a `Dask.dataframe` with a `Dask.dataframe` of a single partition.
3. Joining `Dask.dataframes` along their indexes

It is expensive in the following case:

1. Joining `Dask.dataframes` along columns that are not their index

The expensive case requires a shuffle. This is fine, and `Dask.dataframe` will complete the job well, but it will be more expensive than a typical linear-time operation.

```
dd.merge(a, pandas_df) # fast
dd.merge(a, b, left_index=True, right_index=True) # fast
dd.merge(a, b, left_index=True, right_on='id') # half-fast, half-slow
dd.merge(a, b, left_on='id', right_on='id') # slow
```

## Store Data in Apache Parquet Format

HDF5 is a popular choice for Pandas users with high performance needs. We encourage `Dask.dataframe` users to [store and load data](#) using Parquet instead. [Apache Parquet](#) is a columnar binary format that is easy to split into multiple files (easier for parallel loading) and is generally much simpler to deal with than HDF5 (from the library’s perspective). It is also a common format used by other big data systems like [Apache Spark](#) and [Apache Impala \(incubating\)](#) and so is useful to interchange with other systems.

```
df.to_parquet('path/to/my-results/')
df = dd.read_parquet('path/to/my-results/')
```

Dask supports reading with multiple implementations of the Apache Parquet format for Python.

```
df1 = dd.read_parquet('path/to/my-results/', engine='fastparquet')
df2 = dd.read_parquet('path/to/my-results/', engine='arrow')
```

These libraries be installed using

```
conda install fastparquet pyarrow -c conda-forge
```

Fastparquet is a Python-based implementation that uses the [Numba Python-to-LLVM compiler](#). PyArrow is part of the [Apache Arrow project](#) and uses the [C++ implementation of Apache Parquet](#).

Other topics

## 4.8.5 Internal Design

Dask dataframes coordinate many Pandas DataFrames/Series arranged along an index. We define a `dask.dataframe` object with the following components:

- A dask graph with a special set of keys designating partitions, such as `('x', 0)`, `('x', 1)`, ....
- A name to identify which keys in the dask graph refer to this dataframe, such as `'x'`.
- An empty pandas object containing appropriate metadata (e.g. column names, dtypes, etc...).
- A sequence of partition boundaries along the index, called `divisions`.

### Metadata

Many dataframe operations rely on knowing the name and dtype of columns. To keep track of this information, all `dask.dataframe` objects have a `_meta` attribute which contains an empty pandas object with the same dtypes and names. For example:

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': ['x', 'y', 'z']})
>>> ddf = dd.from_pandas(df, npartitions=2)
>>> ddf._meta
Empty DataFrame
Columns: [a, b]
Index: []
>>> ddf._meta.dtypes
a      int64
b      object
dtype: object
```

Internally `dask.dataframe` does its best to propagate this information through all operations, so most of the time a user shouldn't have to worry about this. Usually this is done by evaluating the operation on a small sample of fake data, which can be found on the `_meta_nonempty` attribute:

```
>>> ddf._meta_nonempty
   a  b
0  1  foo
1  1  foo
```

Sometimes this operation may fail in user defined functions (e.g. when using `DataFrame.apply`), or may be prohibitively expensive. For these cases, many functions support an optional `meta` keyword, which allows specifying the metadata directly, avoiding the inference step. For convenience, this supports several options:

1. A pandas object with appropriate dtypes and names. If not empty, an empty slice will be taken:

```
>>> ddf.map_partitions(foo, meta=pd.DataFrame({'a': [1], 'b': [2]}))
```

2. A description of the appropriate names and dtypes. This can take several forms:

- A dict of `{name: dtype}` or an iterable of `(name, dtype)` specifies a dataframe
- A tuple of `(name, dtype)` specifies a series
- A dtype object or string (e.g. `'f8'`) specifies a scalar

This keyword is available on all functions/methods that take user provided callables (e.g. `DataFrame.map_partitions`, `DataFrame.apply`, etc...), as well as many creation functions (e.g. `dd.from_delayed`).

## Categoricals

Dask dataframe divides [categorical data](#) into two types:

- Known categoricals have the `categories` known statically (on the `_meta` attribute). Each partition **must** have the same categories as found on the `_meta` attribute.
- Unknown categoricals don't know the categories statically, and may have different categories in each partition. Internally, unknown categoricals are indicated by the presence of `dd.utils.UNKNOWN_CATEGORIES` in the categories on the `_meta` attribute. Since most dataframe operations propagate the categories, the known/unknown status should propagate through operations (similar to how NaN propagates).

For metadata specified as a description (option 2 above), unknown categoricals are created.

Certain operations are only available for known categoricals. For example, `df.col.cat.categories` would only work if `df.col` has known categories, since the categorical mapping is only known statically on the metadata of known categoricals.

The known/unknown status for a categorical column can be found using the `known` property on the categorical accessor:

```
>>> ddf.col.cat.known
False
```

Additionally, an unknown categorical can be converted to known using `.cat.as_known()`. If you have multiple categorical columns in a dataframe, you may instead want to use `df.categorize(columns=...)`, which will convert all specified columns to known categoricals. Since getting the categories requires a full scan of the data, using `df.categorize()` is more efficient than calling `.cat.as_known()` for each column (which would result in multiple scans).

```
>>> col_known = ddf.col.cat.as_known() # use for single column
>>> col_known.cat.known
True
>>> ddf_known = ddf.categorize() # use for multiple columns
>>> ddf_known.col.cat.known
True
```

To convert a known categorical to an unknown categorical, there is also the `.cat.as_unknown()` method. This requires no computation, as it's just a change in the metadata.

Non-categorical columns can be converted to categoricals in a few different ways:

```
# astype operates lazily, and results in unknown categoricals
ddf = ddf.astype({'mycol': 'category', ...})
# or
ddf['mycol'] = ddf.mycol.astype('category')

# categorize requires computation, and results in known categoricals
ddf = ddf.categorize(columns=['mycol', ...])
```

Additionally, with pandas 0.19.2 and up `dd.read_csv` and `dd.read_table` can read data directly into unknown categorical columns by specifying a column dtype as 'category':

```
>>> ddf = dd.read_csv(..., dtype={col_name: 'category'})
```

With pandas 0.21.0 and up, `dd.read_csv` and `dd.read_table` can read data directly into *known* categoricals by specifying instances of `pd.api.types.CategoricalDtype`:

```
>>> dtype = {'col': pd.api.types.CategoricalDtype(['a', 'b', 'c'])}
>>> ddf = dd.read_csv(..., dtype=dtype)
```

## 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 with expensive algorithms (e.g. `groupby`'s, `joins`, etc...).

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]
```

In these cases any operation that requires a cleanly partitioned dataframe with known divisions will have to perform a sort. This can generally be achieved by calling `df.set_index(...)`.

### 4.8.6 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, even if partitions are not cleanly divided with known divisions. This is the common case.

Additionally, if divisions are known then applying an arbitrary function to groups is efficient when the grouping columns include the index.

Joins are also quite fast when joining a Dask dataframe to a Pandas dataframe or when joining two Dask dataframes along their index. 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
>>> df.groupby(columns_with_index).apply(user_fn)   # Fast and common case
>>> dask_df.join(pandas_df, on=column)             # Fast and common case
>>> lhs.join(rhs)                                  # Fast and common case
>>> lhs.merge(rhs, on=columns_with_index)          # Fast and common case
```

## Difficult Cases

In some cases, such as when applying an arbitrary function to groups (when not grouping on index with known divisions), 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_no_index).apply(user_fn)    # Requires shuffle
>>> lhs.join(rhs, on=columns_no_index)             # 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.Client` such as would be the case if the user sets the Client as default using one of the following two options:

```
client = Client('scheduler:8786', set_as_default=True)

or

dask.set_options(get=client.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')
```

## 4.8.7 Aggregate

Dask support Pandas' aggregate syntax to run multiple reductions on the same groups. Common reductions, such as `max`, `sum`, `mean` are directly supported:

```
>>> df.groupby(columns).aggregate(['sum', 'mean', 'max', 'min'])
```

Dask also supports user defined reductions. To ensure proper performance, the reduction has to be formulated in terms of three independent steps. The `chunk` step is applied to each partition independently and reduces the data within a partition. The `aggregate` combines the within partition results. The optional `finalize` step combines the results returned from the `aggregate` step and should return a single final column. For Dask to recognize the reduction, it has to be passed as an instance of `dask.dataframe.Aggregation`.

For example, `sum` could be implemented as

```
custom_sum = dd.Aggregation('custom_sum', lambda s: s.sum(), lambda s0: s0.sum())
df.groupby('g').agg(custom_sum)
```

The name argument should be different from existing reductions to avoid data corruption. The arguments to each function are pre-grouped series objects, similar to `df.groupby('g')['value']`.

Many reductions can only be implemented with multiple temporaries. To implement these reductions, the steps should return tuples and expect multiple arguments. A mean function can be implemented as

```
custom_mean = dd.Aggregation(
    'custom_mean',
    lambda s: (s.count(), s.sum()),
    lambda count, sum: (count.sum(), sum.sum()),
    lambda count, sum: sum / count,
)
df.groupby('g').agg(custom_mean)
```

## 4.9 Delayed

Sometimes problems don't fit into one of the collections like `dask.array` or `dask.dataframe`. In these cases, users can parallelize custom algorithms using the simpler `dask.delayed` interface. This allows one to create graphs directly with a light annotation of normal python code.

```
>>> x = dask.delayed(inc)(1)
>>> y = dask.delayed(inc)(2)
>>> z = dask.delayed(add)(x, y)
>>> z.compute()
7
>>> z.vizualize()
```

### 4.9.1 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.

## 4.9.2 API

---

*delayed*

Wraps a function or object to produce a `Delayed`.

---

`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. If not provided, the default is to check the global `delayed_pure` setting, and fallback to `False` if unset.

**nout** : int, optional

The number of outputs returned from calling the resulting `Delayed` object. If provided, the `Delayed` output of the call can be iterated into `nout` objects, allowing for unpacking of results. By default iteration over `Delayed` objects will error. Note, that `nout=1` expects `obj`, to return a tuple of length 1, and consequently for `nout=0`, `obj` should return an empty tuple.

**traverse** : bool, optional

By default dask traverses builtin python collections looking for dask objects passed to `delayed`. For large collections this can be expensive. If `obj` doesn't contain any dask objects, set `traverse=False` to avoid doing this traversal.

### 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`, 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
```

Instead of setting `pure` as a property of the callable, you can also set it contextually using the `delayed_pure` setting. Note that this influences the *call* and not the *creation* of the callable:

```
>>> import dask
>>> @delayed
... def mul(a, b):
...     return a * b
>>> with dask.set_options(delayed_pure=True):
...     print(mul(1, 2).key == mul(1, 2).key)
True
>>> with dask.set_options(delayed_pure=False):
...     print(mul(1, 2).key == mul(1, 2).key)
False
```

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 if `pure=True` or is generated randomly if `pure=False` (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'")
```

“Magic” methods (e.g. operators and attribute access) are assumed to be pure, meaning that subsequent calls must return the same results. This is not overrideable. To invoke an impure attribute or operator, you'd need to use it in a delayed function with `pure=False`.

```
>>> class Incrementer(object):
...     def __init__(self):
...         self._n = 0
...     @property
...     def n(self):
...         self._n += 1
...         return self._n
...
>>> x = delayed(Incrementer())
>>> x.n.key == x.n.key
True
>>> get_n = delayed(lambda x: x.n, pure=False)
>>> get_n(x).key == get_n(x).key
False
```

In contrast, 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 respect the global `delayed_pure` setting and support the `dask_key_name` keyword:

```
>>> a.count(2, dask_key_name="count_2")
Delayed('count_2')
>>> with dask.set_options(delayed_pure=True):
...     print(a.count(2).key == a.count(2).key)
True
```

### 4.9.3 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.

## 4.10 Futures

Dask supports a real-time task framework that extends Python's `concurrent.futures` interface. This interface is good for arbitrary task scheduling, like `dask.delayed`, but is immediate rather than lazy, which provides some more flexibility in situations where the computations may evolve over time.

These features depend on the second generation task scheduler found in `dask.distributed` (which, despite its name, runs very well on a single machine).

### 4.10.1 Start Dask Client

You must start a `Client` to use the futures interface. This tracks state among the various worker processes or threads.

```
from dask.distributed import Client

client = Client() # start local workers as processes
```

```
# or
client = Client(processes=False) # start local workers as threads
```

If you have `Bokeh` installed then this starts up a diagnostic dashboard at <http://localhost:8787>.

## 4.10.2 Submit Tasks

<code>Client.submit(func, *args, **kwargs)</code>	Submit a function application to the scheduler
<code>Client.map(func, *iterables, **kwargs)</code>	Map a function on a sequence of arguments
<code>Future.result([timeout])</code>	Wait until computation completes, gather result to local process.

Then you can submit individual tasks using the `submit` method.

```
def inc(x):
    return x + 1

def add(x, y):
    return x + y

a = client.submit(inc, 10) # calls inc(10) in background thread or process
b = client.submit(inc, 20) # calls inc(20) in background thread or process
```

`submit` returns a `Future`, which refers to a remote result. This result may not yet be completed:

```
>>> a
<Future: status: pending, key: inc-b8aaf26b99466a7a1980efalade6701d>
```

Eventually it will complete. The result stays in the remote thread/process/worker until you ask for it back explicitly.

```
>>> a
<Future: status: finished, type: int, key: inc-b8aaf26b99466a7a1980efalade6701d>

>>> a.result() # blocks until task completes and data arrives
11
```

You can pass futures as inputs to `submit`. Dask automatically handles dependency tracking; once all input futures have completed they will be moved onto a single worker (if necessary), and then the computation that depends on them will be started. You do not need to wait for inputs to finish before submitting a new task; Dask will handle this automatically.

```
c = client.submit(add, a, b) # calls add on the results of a and b
```

Similar to Python's `map` you can use `Client.map` to call the same function and many inputs:

```
futures = client.map(inc, range(1000))
```

However note that each task comes with about 1ms of overhead. If you want to map a function over a large number of inputs then you might consider `dask.bag` or `dask.dataframe` instead.

## 4.10.3 Move Data

<code>Future.result([timeout])</code>	Wait until computation completes, gather result to local process.
<code>Client.gather(futures[, errors, maxsize, ...])</code>	Gather futures from distributed memory
<code>Client.scatter(data[, workers, broadcast, ...])</code>	Scatter data into distributed memory

Given any future you can call the `.result` method to gather the result. This will block until the future is done computing and then transfer the result back to your local process if necessary.

```
>>> c.result()
32
```

You can gather many results concurrently using the `Client.gather` method. This can be more efficient than calling `.result()` on each future sequentially.

```
>>> # results = [future.result() for future in futures]
>>> results = client.gather(futures) # this can be faster
```

If you have important local data that you want to include in your computation you can either include it as a normal input to a `submit` or `map` call:

```
>>> df = pd.read_csv('training-data.csv')
>>> future = client.submit(my_function, df)
```

Or you can `scatter` it explicitly. Scattering moves your data to a worker and returns a future pointing to that data:

```
>>> remote_df = client.scatter(df)
>>> remote_df
<Future: status: finished, type: DataFrame, key: bbd0ca93589c56ea14af49cba470006e>

>>> future = client.submit(my_function, remote_df)
```

Both of these accomplish the same result, but using `scatter` can sometimes be faster. This is especially true if you use processes or distributed workers (where data transfer is necessary) and you want to use `df` in many computations. Scattering the data beforehand avoids excessive data movement.

Calling `scatter` on a list scatters all elements individually. Dask will spread these elements evenly throughout workers in a round-robin fashion:

```
>>> client.scatter([1, 2, 3])
[<Future: status: finished, type: int, key: c0a8a20f903a4915b94db8de3ea63195>,
 <Future: status: finished, type: int, key: 58e78e1b34eb49a68c65b54815d1b158>,
 <Future: status: finished, type: int, key: d3395e15f605bc35ab1bac6341a285e2>]
```

#### 4.10.4 References, Cancellation, and Exceptions

<code>Future.cancel(**kwargs)</code>	Cancel request to run this future
<code>Future.exception([timeout])</code>	Return the exception of a failed task
<code>Future.traceback([timeout])</code>	Return the traceback of a failed task
<code>Client.cancel(futures[, asynchronous, force])</code>	Cancel running futures

Dask will only compute and hold onto results for which there are active futures. In this way your local variables define what is active in Dask. When a future is garbage collected by your local Python session, Dask will feel free to delete that data or stop ongoing computations that were trying to produce it.

```
>>> del future # deletes remote data once future is garbage collected
```

You can also explicitly cancel a task using the `Future.cancel` or `Client.cancel` methods.

```
>>> future.cancel() # deletes data even if other futures point to it
```

If a future fails, then Dask will raise the remote exceptions and tracebacks if you try to get the result.

```
def div(x, y):
    return x / y

>>> a = client.submit(div, 1, 0) # 1 / 0 raises a ZeroDivisionError
>>> a
<Future: status: error, key: div-3601743182196fb56339e584a2bf1039>

>>> a.result()
      1 def div(x, y):
----> 2     return x / y

ZeroDivisionError: division by zero
```

All futures that depend on an erred future also err with the same exception:

```
>>> b = client.submit(inc, a)
>>> b
<Future: status: error, key: inc-15e2e4450a0227fa38ede4d6b1a952db>
```

You can collect the exception or traceback explicitly with the `Future.exception` or `Future.traceback` methods.

### 4.10.5 Waiting on Futures

<code>as_completed([futures, loop, with_results])</code>	Return futures in the order in which they complete
<code>wait(fs[, timeout, return_when])</code>	Wait until all futures are complete

You can wait on a future or collection of futures using the `wait` function:

```
from dask.distributed import wait

>>> wait(futures)
```

This blocks until all futures are finished or have erred.

You can also iterate over the futures as they complete using the `as_completed` function:

```
from dask.distributed import as_completed

futures = client.map(score, x_values)

best = -1
for future in as_completed(futures):
    y = future.result()
    if y > best:
        best = y
```

For greater efficiency you can also ask `as_completed` to gather the results in the background.

```
for future in as_completed(futures, results=True):
    ...
```

Or collect futures all futures in batches that had arrived since the last iteration

```
for batch in as_completed(futures, results=True).batches():
    for future in batch:
        ...
```

Additionally, for iterative algorithms you can add more futures into the `as_completed` iterator

```
seq = as_completed(futures)

for future in seq:
    y = future.result()
    if condition(y):
        new_future = client.submit(...)
        seq.add(new_future) # add back into the loop
```

## 4.10.6 Fire and Forget

---

`fire_and_forget(obj)`

Run tasks at least once, even if we release the futures

---

Sometimes we don't care about gathering the result of a task, and only care about side effects that it might have, like writing a result to a file.

```
>>> a = client.submit(load, filename)
>>> b = client.submit(process, a)
>>> c = client.submit(write, c, out_filename)
```

As noted above, Dask will stop work that doesn't have any active futures. It thinks that because no one has a pointer to this data that no one cares. You can tell Dask to compute a task anyway, even if there are no active futures, using the `fire_and_forget` function:

```
from dask.distributed import fire_and_forget

>>> fire_and_forget(c)
```

This is particularly useful when a future may go out of scope, for example as part of a function:

```
def process(filename):
    out_filename = 'out-' + filename
    a = client.submit(load, filename)
    b = client.submit(process, a)
    c = client.submit(write, c, out_filename)
    fire_and_forget(c)
    return # here we lose the reference to c, but that's now ok

for filename in filenames:
    process(filename)
```

## 4.10.7 Submit Tasks from Tasks

<code>get_client([address, timeout])</code>	Get a client while within a task
<code>secede()</code>	Have this task secede from the worker's thread pool

Tasks can launch other tasks by getting their own client. This enables complex and highly dynamic workloads.

```
from dask.distributed import get_client

def my_function(x):
    ...

    # Get locally created client
    client = get_client()

    # Do normal client operations, asking cluster for computation
    a = client.submit(...)
    b = client.submit(...)
    a, b = client.gather([a, b])

    return a + b
```

It also allows you to set up long running tasks that watch other resources like sockets or physical sensors:

```
def monitor(device):
    client = get_client()
    while True:
        data = device.read_data()
        future = client.submit(process, data)
        fire_and_forget(future)

for device in devices:
    fire_and_forget(client.submit(monitor))
```

However, each running task takes up a single thread, and so if you launch many tasks that launch other tasks then it is possible to deadlock the system if you are not careful. You can call the `secede` function from within a task to have it remove itself from the dedicated thread pool into an administrative thread that does not take up a slot within the Dask worker:

```
from dask.distributed import get_client, secede

def monitor(device):
    client = get_client()
    secede()
    while True:
        data = device.read_data()
        future = client.submit(process, data)
        fire_and_forget(future)
```

## 4.10.8 Coordinate Data Between Clients

<code>Queue([name, client, maxsize])</code>	Distributed Queue
<code>Variable([name, client, maxsize])</code>	Distributed Global Variable

In the section above we saw that you could have multiple clients running at the same time, each of which generated and manipulated futures. These clients can coordinate with each other using `Dask Queue` and `Variable` objects, which can communicate futures or small bits of data between clients sensibly.

Dask queues follow the API for the standard Python Queue, but now move futures or small messages between clients. Queues serialize sensibly and reconnect themselves on remote clients if necessary.

```
from dask.distributed import Queue

def load_and_submit(filename):
    data = load(filename)
    client = get_client()
    future = client.submit(process, data)
    queue.put(future)

client = Client()

queue = Queue()

for filename in filenames:
    future = client.submit(load_and_submit, filename)
    fire_and_forget(future)

while True:
    future = queue.get()
    print(future.result())
```

Queues can also send small pieces of information, anything that is msgpack encodable (ints, strings, bools, lists, dicts, etc..). This can be useful to send back small scores or administrative messages:

```
def func(x):
    try:
        ...
    except Exception as e:
        error_queue.put(str(e))

error_queue = Queue()
```

Variables are like Queues in that they communicate futures and small data between clients. However variables hold only a single value. You can get or set that value at any time.

```
>>> var = Variable('stopping-criterion')
>>> var.set(False)

>>> var.get()
False
```

This is often used to signal stopping criteria or current parameters, etc. between clients.

If you want to share large pieces of information then scatter the data first

```
>>> parameters = np.array(...)
>>> future = client.scatter(parameters)
>>> var.set(future)
```

## 4.10.9 API

### Client

<code>Client([address, loop, timeout, ...])</code>	Connect to and drive computation on a distributed Dask cluster
<code>Client.cancel(futures[, asynchronous, force])</code>	Cancel running futures
<code>Client.compute(collections[, sync, ...])</code>	Compute dask collections on cluster
<code>Client.gather(futures[, errors, maxsize, ...])</code>	Gather futures from distributed memory
<code>Client.get(dsk, keys[, restrictions, ...])</code>	Compute dask graph
<code>Client.get_dataset(name, **kwargs)</code>	Get named dataset from the scheduler
<code>Client.get_executor(**kwargs)</code>	Return a concurrent.futures Executor for submitting tasks on this Client.
<code>Client.has_what([workers])</code>	Which keys are held by which workers
<code>Client.list_datasets(**kwargs)</code>	List named datasets available on the scheduler
<code>Client.map(func, *iterables, **kwargs)</code>	Map a function on a sequence of arguments
<code>Client.ncores([workers])</code>	The number of threads/cores available on each worker node
<code>Client.persist(collections[, ...])</code>	Persist dask collections on cluster
<code>Client.publish_dataset(**kwargs)</code>	Publish named datasets to scheduler
<code>Client.rebalance([futures, workers])</code>	Rebalance data within network
<code>Client.replicate(futures[, n, workers, ...])</code>	Set replication of futures within network
<code>Client.restart(**kwargs)</code>	Restart the distributed network
<code>Client.run(function, *args, **kwargs)</code>	Run a function on all workers outside of task scheduling system
<code>Client.run_on_scheduler(function, *args, ...)</code>	Run a function on the scheduler process
<code>Client.scatter(data[, workers, broadcast, ...])</code>	Scatter data into distributed memory
<code>Client.shutdown([timeout])</code>	Close this client
<code>Client.scheduler_info(**kwargs)</code>	Basic information about the workers in the cluster
<code>Client.shutdown([timeout])</code>	Close this client
<code>Client.start_ipython_workers([workers, ...])</code>	Start IPython kernels on workers
<code>Client.start_ipython_scheduler([magic_name, ...])</code>	Start IPython kernel on the scheduler
<code>Client.submit(func, *args, **kwargs)</code>	Submit a function application to the scheduler
<code>Client.unpublish_dataset(name, **kwargs)</code>	Remove named datasets from scheduler
<code>Client.upload_file(filename, **kwargs)</code>	Upload local package to workers
<code>Client.who_has([futures])</code>	The workers storing each future's data

### Future

<code>Future(key[, client, inform, state])</code>	A remotely running computation
<code>Future.add_done_callback(fn)</code>	Call callback on future when callback has finished
<code>Future.cancel(**kwargs)</code>	Cancel request to run this future
<code>Future.cancelled()</code>	Returns True if the future has been cancelled
<code>Future.done()</code>	Is the computation complete?
<code>Future.exception([timeout])</code>	Return the exception of a failed task
<code>Future.result([timeout])</code>	Wait until computation completes, gather result to local process.
<code>Future.traceback([timeout])</code>	Return the traceback of a failed task

### Functions

<code>as_completed([futures, loop, with_results])</code>	Return futures in the order in which they complete
<code>fire_and_forget(obj)</code>	Run tasks at least once, even if we release the futures
<code>get_client([address, timeout])</code>	Get a client while within a task
<code>secede()</code>	Have this task secede from the worker's thread pool
<code>wait(fs[, timeout, return_when])</code>	Wait until all futures are complete

`distributed.as_completed` (*futures=None, loop=None, with\_results=False*)

Return futures in the order in which they complete

This returns an iterator that yields the input future objects in the order in which they complete. Calling `next` on the iterator will block until the next future completes, irrespective of order.

Additionally, you can also add more futures to this object during computation with the `.add` method

### Examples

```
>>> x, y, z = client.map(inc, [1, 2, 3])
>>> for future in as_completed([x, y, z]):
...     print(future.result())
3
2
4
```

Add more futures during computation

```
>>> x, y, z = client.map(inc, [1, 2, 3])
>>> ac = as_completed([x, y, z])
>>> for future in ac:
...     print(future.result())
...     if random.random() < 0.5:
...         ac.add(c.submit(double, future))
4
2
8
3
6
12
24
```

Optionally wait until the result has been gathered as well

```
>>> ac = as_completed([x, y, z], with_results=True)
>>> for future, result in ac:
...     print(result)
2
4
3
```

`distributed.fire_and_forget` (*obj*)

Run tasks at least once, even if we release the futures

Under normal operation Dask will not run any tasks for which there is not an active future (this avoids unnecessary work in many situations). However sometimes you want to just fire off a task, not track its future, and expect it to finish eventually. You can use this function on a future or collection of futures to ask Dask to complete the task even if no active client is tracking it.

The results will not be kept in memory after the task completes (unless there is an active future) so this is only useful for tasks that depend on side effects.

**Parameters** **obj**: Future, list, dict, dask collection

The futures that you want to run at least once

### Examples

```
>>> fire_and_forget(client.submit(func, *args))
```

`distributed.get_client` (*address=None, timeout=3*)

Get a client while within a task

This client connects to the same scheduler to which the worker is connected

**See also:**

`get_worker`, `worker_client`, `secede`

### Examples

```
>>> def f():
...     client = get_client()
...     futures = client.map(lambda x: x + 1, range(10)) # spawn many tasks
...     results = client.gather(futures)
...     return sum(results)
```

```
>>> future = client.submit(f)
>>> future.result()
55
```

`distributed.secede` ()

Have this task secede from the worker's thread pool

This opens up a new scheduling slot and a new thread for a new task. This enables the client to schedule tasks on this node, which is especially useful while waiting for other jobs to finish (e.g., with `client.gather`).

**See also:**

`get_client`, `get_worker`

### Examples

```
>>> def mytask(x):
...     # do some work
...     client = get_client()
...     futures = client.map(...) # do some remote work
...     secede() # while that work happens, remove ourself from the pool
...     return client.gather(futures) # return gathered results
```

`distributed.wait` (*fs, timeout=None, return\_when='ALL\_COMPLETED'*)

Wait until all futures are complete

**Parameters fs: list of futures****timeout: number, optional**

Time in seconds after which to raise a `gen.TimeoutError`

**Returns** Named tuple of completed, not completed

```
class distributed.Client (address=None, loop=None, timeout=5, set_as_default=True, scheduler_file=None, security=None, asynchronous=False, name=None, **kwargs)
```

Connect to and drive computation on a distributed Dask cluster

The Client connects users to a `dask.distributed` compute cluster. It provides an asynchronous user interface around functions and futures. This class resembles executors in `concurrent.futures` but also allows Future objects within `submit/map` calls.

**Parameters address: string, or Cluster**

This can be the address of a Scheduler server like a string `'127.0.0.1:8786'` or a cluster object like `LocalCluster()`

**timeout: int**

Timeout duration for initial connection to the scheduler

**set\_as\_default: bool (True)**

Claim this scheduler as the global dask scheduler

**scheduler\_file: string (optional)**

Path to a file with scheduler information if available

**security: (optional)**

Optional security information

**asynchronous: bool (False by default)**

Set to True if this client will be used within a Tornado event loop

**name: string (optional)**

Gives the client a name that will be included in logs generated on the scheduler for matters relating to this client

See also:

**distributed.scheduler.Scheduler** Internal scheduler

## Examples

Provide cluster's scheduler node address on initialization:

```
>>> client = Client('127.0.0.1:8786')
```

Use `submit` method to send individual computations to the cluster

```
>>> a = client.submit(add, 1, 2)
>>> b = client.submit(add, 10, 20)
```

Continue using `submit` or `map` on results to build up larger computations

```
>>> c = client.submit(add, a, b)
```

Gather results with the `gather` method.

```
>>> client.gather(c)
33
```

### asynchronous

Are we running in the event loop?

This is true if the user signaled that we might be when creating the client as in the following:

```
client = Client(asynchronous=True)
```

However, we override this expectation if we can definitively tell that we are running from a thread that is not the event loop. This is common when calling `get_client()` from within a worker task. Even though the client was originally created in asynchronous mode we may find ourselves in contexts when it is better to operate synchronously.

### call\_stack (futures=None, keys=None)

The actively running call stack of all relevant keys

You can specify data of interest either by providing futures or collections in the `futures=` keyword or a list of explicit keys in the `keys=` keyword. If neither are provided then all call stacks will be returned.

#### Parameters futures: list (optional)

List of futures, defaults to all data

#### keys: list (optional)

List of key names, defaults to all data

### Examples

```
>>> df = dd.read_parquet(...).persist()
>>> client.call_stack(df) # call on collections
```

```
>>> client.call_stack() # Or call with no arguments for all activity
```

### cancel (futures, asynchronous=None, force=False)

Cancel running futures

This stops future tasks from being scheduled if they have not yet run and deletes them if they have already run. After calling, this result and all dependent results will no longer be accessible

#### Parameters futures: list of Futures

#### force: boolean (False)

Cancel this future even if other clients desire it

### close (timeout=10)

Close this client

Clients will also close automatically when your Python session ends

If you started a client without arguments like `Client()` then this will also close the local cluster that was started at the same time.

**See also:**

`Client.restart`

**compute** (*collections, sync=False, optimize\_graph=True, workers=None, allow\_other\_workers=False, resources=None, retries=0, \*\*kwargs*)  
Compute dask collections on cluster

**Parameters collections: iterable of dask objects or single dask object**

Collections like `dask.array` or `dataframe` or `dask.value` objects

**sync: bool (optional)**

Returns Futures if False (default) or concrete values if True

**optimize\_graph: bool**

Whether or not to optimize the underlying graphs

**workers: str, list, dict**

Which workers can run which parts of the computation If a string a list then the output collections will run on the listed

workers, but other sub-computations can run anywhere

**If a dict then keys should be (tuples of) collections and values** should be addresses or lists.

**allow\_other\_workers: bool, list**

If True then all restrictions in `workers=` are considered loose If a list then only the keys for the listed collections are loose

**retries: int (default to 0)**

Number of allowed automatic retries if computing a result fails

**\*\*kwargs:**

Options to pass to the graph optimize calls

**Returns** List of Futures if input is a sequence, or a single future otherwise

**See also:**

`Client.get` Normal synchronous `dask.get` function

**Examples**

```
>>> from dask import delayed
>>> from operator import add
>>> x = delayed(add)(1, 2)
>>> y = delayed(add)(x, x)
>>> xx, yy = client.compute([x, y])
>>> xx
<Future: status: finished, key: add-8f6e709446674bad78ea8aeecfee188e>
>>> xx.result()
3
>>> yy.result()
6
```

Also support single arguments

```
>>> xx = client.compute(x)
```

**classmethod** `current()`

Return global client if one exists, otherwise raise ValueError

**gather** (*futures*, *errors='raise'*, *maxsize=0*, *direct=None*, *asynchronous=None*)

Gather futures from distributed memory

Accepts a future, nested container of futures, iterator, or queue. The return type will match the input type.

**Parameters** **futures: Collection of futures**

This can be a possibly nested collection of Future objects. Collections can be lists, sets, iterators, queues or dictionaries

**errors: string**

Either 'raise' or 'skip' if we should raise if a future has erred or skip its inclusion in the output collection

**maxsize: int**

If the input is a queue then this produces an output queue with a maximum size.

**Returns** results: a collection of the same type as the input, but now with

gathered results rather than futures

**See also:**

`Client.scatter` Send data out to cluster

## Examples

```
>>> from operator import add
>>> c = Client('127.0.0.1:8787')
>>> x = c.submit(add, 1, 2)
>>> c.gather(x)
3
>>> c.gather([x, [x], x]) # support lists and dicts
[3, [3], 3]
```

```
>>> seq = c.gather(iter([x, x])) # support iterators
>>> next(seq)
3
```

**get** (*dsk*, *keys*, *restrictions=None*, *loose\_restrictions=None*, *resources=None*, *sync=True*, *asynchronous=None*, *\*\*kwargs*)

Compute dask graph

**Parameters** **dsk: dict**

**keys: object, or nested lists of objects**

**restrictions: dict (optional)**

A mapping of {key: {set of worker hostnames}} that restricts where jobs can take place

**sync: bool (optional)**

Returns Futures if False or concrete values if True (default).

**See also:**

`Client.compute` Compute asynchronous collections

### Examples

```
>>> from operator import add
>>> c = Client('127.0.0.1:8787')
>>> c.get({'x': (add, 1, 2)}, 'x')
3
```

**get\_dataset** (*name*, *\*\*kwargs*)

Get named dataset from the scheduler

**See also:**

`Client.publish_dataset`, `Client.list_datasets`

**get\_executor** (*\*\*kwargs*)

Return a concurrent.futures Executor for submitting tasks on this Client.

**Parameters** *\*\*kwargs*:

Any submit()- or map()- compatible arguments, such as *workers* or *resources*.

**Returns** An Executor object that's fully compatible with the concurrent.futures

API.

**get\_metadata** (*keys*, *default='\_\_no\_default\_\_'*)

Get arbitrary metadata from scheduler

See set\_metadata for the full docstring with examples

**See also:**

`Client.set_metadata`

**classmethod get\_restrictions** (*collections*, *workers*, *allow\_other\_workers*)

Get restrictions from inputs to compute/persist

**get\_scheduler\_logs** (*n=None*)

Get logs from scheduler

**Parameters** *n*: int

Number of logs to retrieve. Maxes out at 10000 by default, configurable in config.yaml::log-length

**Returns** Logs in reversed order (newest first)

**get\_versions** (*check=False*)

Return version info for the scheduler, all workers and myself

**Parameters** *check* : boolean, default False

raise ValueError if all required & optional packages do not match

## Examples

```
>>> c.get_versions()
```

**get\_worker\_logs** (*n=None, workers=None*)

Get logs from workers

**Parameters** **n: int**

Number of logs to retrieve. Maxes out at 10000 by default, configurable in `config.yaml::log-length`

**workers: iterable**

List of worker addresses to retrieve. Gets all workers by default.

**Returns** Dictionary mapping worker address to logs.

Logs are returned in reversed order (newest first)

**has\_what** (*workers=None, \*\*kwargs*)

Which keys are held by which workers

**Parameters** **workers: list (optional)**

A list of worker addresses, defaults to all

**See also:**

`Client.who_has`, `Client.ncores`

## Examples

```
>>> x, y, z = c.map(inc, [1, 2, 3])
>>> wait([x, y, z])
>>> c.has_what()
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
                       'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
                       'inc-1e297fc27658d7b67b3a758f16bcf47a']}
```

**list\_datasets** (*\*\*kwargs*)

List named datasets available on the scheduler

**See also:**

`Client.publish_dataset`, `Client.get_dataset`

**map** (*func, \*iterables, \*\*kwargs*)

Map a function on a sequence of arguments

Arguments can be normal objects or Futures

**Parameters** **func: callable**

**iterables: Iterables, Iterators, or Queues**

**key: str, list**

Prefix for task names if string. Explicit names if list.

**pure: bool (defaults to True)**

Whether or not the function is pure. Set `pure=False` for impure functions like `np.random.random`.

**workers: set, iterable of sets**

A set of worker hostnames on which computations may be performed. Leave empty to default to all workers (common case)

**retries: int (default to 0)**

Number of allowed automatic retries if a task fails

**Returns** List, iterator, or Queue of futures, depending on the type of the inputs.

**See also:**

`Client.submit` Submit a single function

### Examples

```
>>> L = client.map(func, sequence)
```

**nbytes** (*keys=None, summary=True, \*\*kwargs*)

The bytes taken up by each key on the cluster

This is as measured by `sys.getsizeof` which may not accurately reflect the true cost.

**Parameters keys: list (optional)**

A list of keys, defaults to all keys

**summary: boolean, (optional)**

Summarize keys into key types

**See also:**

`Client.who_has`

### Examples

```
>>> x, y, z = c.map(inc, [1, 2, 3])
>>> c.nbytes(summary=False)
{'inc-1c8dd6be1c21646c71f76c16d09304ea': 28,
 'inc-1e297fc27658d7b67b3a758f16bcf47a': 28,
 'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b': 28}
```

```
>>> c.nbytes(summary=True)
{'inc': 84}
```

**ncores** (*workers=None, \*\*kwargs*)

The number of threads/cores available on each worker node

**Parameters workers: list (optional)**

A list of workers that we care about specifically. Leave empty to receive information about all workers.

**See also:***Client.who\_has, Client.has\_what***Examples**

```
>>> c.ncores()
{'192.168.1.141:46784': 8,
 '192.167.1.142:47548': 8,
 '192.167.1.143:47329': 8,
 '192.167.1.144:37297': 8}
```

**normalize\_collection** (*collection*)

Replace collection's tasks by already existing futures if they exist

This normalizes the tasks within a collections task graph against the known futures within the scheduler. It returns a copy of the collection with a task graph that includes the overlapping futures.

**See also:***Client.persist* trigger computation of collection's tasks**Examples**

```
>>> len(x.__dask_graph__()) # x is a dask collection with 100 tasks
100
>>> set(client.futures).intersection(x.__dask_graph__()) # some overlap_
↳exists
10
```

```
>>> x = client.normalize_collection(x)
>>> len(x.__dask_graph__()) # smaller computational graph
20
```

**persist** (*collections, optimize\_graph=True, workers=None, allow\_other\_workers=None, re-sources=None, retries=None, \*\*kwargs*)

Persist dask collections on cluster

Starts computation of the collection on the cluster in the background. Provides a new dask collection that is semantically identical to the previous one, but now based off of futures currently in execution.

**Parameters collections: sequence or single dask object**

Collections like `dask.array` or `dataframe` or `dask.value` objects

**optimize\_graph: bool**

Whether or not to optimize the underlying graphs

**workers: str, list, dict**

Which workers can run which parts of the computation If a string a list then the output collections will run on the listed

workers, but other sub-computations can run anywhere

**If a dict then keys should be (tuples of) collections and values should be addresses or lists.**

**allow\_other\_workers: bool, list**

If True then all restrictions in workers= are considered loose If a list then only the keys for the listed collections are loose

**retries: int (default to 0)**

Number of allowed automatic retries if computing a result fails

**kwargs:**

Options to pass to the graph optimize calls

**Returns** List of collections, or single collection, depending on type of input.

**See also:**

*Client.compute*

**Examples**

```
>>> xx = client.persist(x)
>>> xx, yy = client.persist([x, y])
```

**processing** (*workers=None*)

The tasks currently running on each worker

**Parameters workers: list (optional)**

A list of worker addresses, defaults to all

**See also:**

*Client.stacks, Client.who\_has, Client.has\_what, Client.ncores*

**Examples**

```
>>> x, y, z = c.map(inc, [1, 2, 3])
>>> c.processing()
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
                      'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
                      'inc-1e297fc27658d7b67b3a758f16bcf47a']}
```

**profile** (*key=None, start=None, stop=None, workers=None, merge\_workers=True*)

Collect statistical profiling information about recent work

**Parameters key: str**

Key prefix to select, this is typically a function name like 'inc' Leave as None to collect all data

**start: time****stop: time****workers: list**

List of workers to restrict profile information

## Examples

```
>>> client.profile() # call on collections
```

### **publish\_dataset** (\*\*kwargs)

Publish named datasets to scheduler

This stores a named reference to a dask collection or list of futures on the scheduler. These references are available to other Clients which can download the collection or futures with `get_dataset`.

Datasets are not immediately computed. You may wish to call `Client.persist` prior to publishing a dataset.

#### **Parameters** kwargs: dict

named collections to publish on the scheduler

#### **Returns** None

#### **See also:**

`Client.list_datasets`, `Client.get_dataset`, `Client.unpublish_dataset`, `Client.persist`

## Examples

Publishing client:

```
>>> df = dd.read_csv('s3://...')
>>> df = c.persist(df)
>>> c.publish_dataset(my_dataset=df)
```

Receiving client:

```
>>> c.list_datasets()
['my_dataset']
>>> df2 = c.get_dataset('my_dataset')
```

### **rebalance** (futures=None, workers=None, \*\*kwargs)

Rebalance data within network

Move data between workers to roughly balance memory burden. This either affects a subset of the keys/workers or the entire network, depending on keyword arguments.

This operation is generally not well tested against normal operation of the scheduler. It is not recommended to use it while waiting on computations.

#### **Parameters** futures: list, optional

A list of futures to balance, defaults all data

#### **workers: list, optional**

A list of workers on which to balance, defaults to all workers

### **replicate** (futures, n=None, workers=None, branching\_factor=2, \*\*kwargs)

Set replication of futures within network

Copy data onto many workers. This helps to broadcast frequently accessed data and it helps to improve resilience.

This performs a tree copy of the data throughout the network individually on each piece of data. This operation blocks until complete. It does not guarantee replication of data to future workers.

**Parameters** **futures:** list of futures

Futures we wish to replicate

**n:** int, optional

Number of processes on the cluster on which to replicate the data. Defaults to all.

**workers:** list of worker addresses

Workers on which we want to restrict the replication. Defaults to all.

**branching\_factor:** int, optional

The number of workers that can copy data in each generation

**See also:**

`Client.rebalance`

## Examples

```
>>> x = c.submit(func, *args)
>>> c.replicate([x]) # send to all workers
>>> c.replicate([x], n=3) # send to three workers
>>> c.replicate([x], workers=['alice', 'bob']) # send to specific
>>> c.replicate([x], n=1, workers=['alice', 'bob']) # send to one of
↳specific workers
>>> c.replicate([x], n=1) # reduce replications
```

**restart** (\*\*kwargs)

Restart the distributed network

This kills all active work, deletes all data on the network, and restarts the worker processes.

**run** (function, \*args, \*\*kwargs)

Run a function on all workers outside of task scheduling system

This calls a function on all currently known workers immediately, blocks until those results come back, and returns the results asynchronously as a dictionary keyed by worker address. This method is generally used for side effects, such as collecting diagnostic information or installing libraries.

If your function takes an input argument named `dask_worker` then that variable will be populated with the worker itself.

**Parameters** **function:** callable

**\*args:** arguments for remote function

**\*\*kwargs:** keyword arguments for remote function

**workers:** list

Workers on which to run the function. Defaults to all known workers.

## Examples

```
>>> c.run(os.getpid)
{'192.168.0.100:9000': 1234,
 '192.168.0.101:9000': 4321,
 '192.168.0.102:9000': 5555}
```

Restrict computation to particular workers with the `workers=` keyword argument.

```
>>> c.run(os.getpid, workers=['192.168.0.100:9000',
...                           '192.168.0.101:9000'])
{'192.168.0.100:9000': 1234,
 '192.168.0.101:9000': 4321}
```

```
>>> def get_status(dask_worker):
...     return dask_worker.status
```

```
>>> c.run(get_hostname)
{'192.168.0.100:9000': 'running',
 '192.168.0.101:9000': 'running'}
```

**run\_coroutine** (*function*, \*args, \*\*kwargs)

Spawn a coroutine on all workers.

This spawns a coroutine on all currently known workers and then waits for the coroutine on each worker. The coroutines' results are returned as a dictionary keyed by worker address.

**Parameters** **function: a coroutine function**

(typically a function wrapped in `gen.coroutine` or a Python 3.5+ async function)

**\*args: arguments for remote function**

**\*\*kwargs: keyword arguments for remote function**

**wait: boolean (default True)**

Whether to wait for coroutines to end.

**workers: list**

Workers on which to run the function. Defaults to all known workers.

**run\_on\_scheduler** (*function*, \*args, \*\*kwargs)

Run a function on the scheduler process

This is typically used for live debugging. The function should take a keyword argument `dask_scheduler=`, which will be given the scheduler object itself.

**See also:**

*Client.run* Run a function on all workers

*Client.start\_ipython\_scheduler* Start an IPython session on scheduler

## Examples

```
>>> def get_number_of_tasks(dask_scheduler=None):  
...     return len(dask_scheduler.task_state)
```

```
>>> client.run_on_scheduler(get_number_of_tasks)  
100
```

**scatter** (*data*, *workers=None*, *broadcast=False*, *direct=None*, *hash=True*, *maxsize=0*, *timeout=3*, *asynchronous=None*)  
Scatter data into distributed memory

This moves data from the local client process into the workers of the distributed scheduler. Note that it is often better to submit jobs to your workers to have them load the data rather than loading data locally and then scattering it out to them.

### Parameters **data**: list, iterator, dict, Queue, or object

Data to scatter out to workers. Output type matches input type.

### **workers**: list of tuples (optional)

Optionally constrain locations of data. Specify workers as hostname/port pairs, e.g. ('127.0.0.1', 8787).

### **broadcast**: bool (defaults to False)

Whether to send each data element to all workers. By default we round-robin based on number of cores.

### **direct**: bool (defaults to automatically check)

Send data directly to workers, bypassing the central scheduler This avoids burdening the scheduler but assumes that the client is able to talk directly with the workers.

### **maxsize**: int (optional)

Maximum size of queue if using queues, 0 implies infinite

### **hash**: bool (optional)

Whether or not to hash data to determine key. If False then this uses a random key

**Returns** List, dict, iterator, or queue of futures matching the type of input.

**See also:**

[\*Client.gather\*](#) Gather data back to local process

## Examples

```
>>> c = Client('127.0.0.1:8787')  
>>> c.scatter(1)  
<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>
```

```
>>> c.scatter([1, 2, 3])
[<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>,
 <Future: status: finished, key: 58e78e1b34eb49a68c65b54815d1b158>,
 <Future: status: finished, key: d3395e15f605bc35ab1bac6341a285e2>]
```

```
>>> c.scatter({'x': 1, 'y': 2, 'z': 3})
{'x': <Future: status: finished, key: x>,
 'y': <Future: status: finished, key: y>,
 'z': <Future: status: finished, key: z>}
```

Constrain location of data to subset of workers

```
>>> c.scatter([1, 2, 3], workers=[('hostname', 8788)])
```

Handle streaming sequences of data with iterators or queues

```
>>> seq = c.scatter(iter([1, 2, 3]))
>>> next(seq)
<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>
```

Broadcast data to all workers

```
>>> [future] = c.scatter([element], broadcast=True)
```

**scheduler\_info** (*\*\*kwargs*)

Basic information about the workers in the cluster

## Examples

```
>>> c.scheduler_info()
{'id': '2de2b6da-69ee-11e6-ab6a-e82aea155996',
 'services': {},
 'type': 'Scheduler',
 'workers': {'127.0.0.1:40575': {'active': 0,
                                'last-seen': 1472038237.4845693,
                                'name': '127.0.0.1:40575',
                                'services': {},
                                'stored': 0,
                                'time-delay': 0.0061032772064208984}}}
```

**set\_metadata** (*key, value*)

Set arbitrary metadata in the scheduler

This allows you to store small amounts of data on the central scheduler process for administrative purposes. Data should be msgpack serializable (ints, strings, lists, dicts)

If the key corresponds to a task then that key will be cleaned up when the task is forgotten by the scheduler.

If the key is a list then it will be assumed that you want to index into a nested dictionary structure using those keys. For example if you call the following:

```
>>> client.set_metadata(['a', 'b', 'c'], 123)
```

Then this is the same as setting

```
>>> scheduler.task_metadata['a']['b']['c'] = 123
```

The lower level dictionaries will be created on demand.

**See also:**

*get\_metadata*

**Examples**

```
>>> client.set_metadata('x', 123)
>>> client.get_metadata('x')
123
```

```
>>> client.set_metadata(['x', 'y'], 123)
>>> client.get_metadata('x')
{'y': 123}
```

```
>>> client.set_metadata(['x', 'w', 'z'], 456)
>>> client.get_metadata('x')
{'y': 123, 'w': {'z': 456}}
```

```
>>> client.get_metadata(['x', 'w'])
{'z': 456}
```

**shutdown** (*timeout=10*)

Close this client

Clients will also close automatically when your Python session ends

If you started a client without arguments like `Client()` then this will also close the local cluster that was started at the same time.

**See also:**

*Client.restart*

**stacks** (*workers=None*)

The task queues on each worker

**Parameters** **workers:** list (optional)

A list of worker addresses, defaults to all

**See also:**

*Client.processing, Client.who\_has, Client.has\_what, Client.ncores*

**Examples**

```
>>> x, y, z = c.map(inc, [1, 2, 3])
>>> c.stacks()
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
                      'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
                      'inc-1e297fc27658d7b67b3a758f16bcf47a']}
```

**start** (*\*\*kwargs*)

Start scheduler running in separate thread

**start\_ipython\_scheduler** (*magic\_name='scheduler\_if\_ipython', qtconsole=False, qtconsole\_args=None*)

Start IPython kernel on the scheduler

**Parameters** **magic\_name: str or None (optional)**

If defined, register IPython magic with this name for executing code on the scheduler. If not defined, register `%scheduler` magic if IPython is running.

**qtconsole: bool (optional)**

If True, launch a Jupyter QtConsole connected to the worker(s).

**qtconsole\_args: list(str) (optional)**

Additional arguments to pass to the qtconsole on startup.

**Returns** `connection_info`: dict

`connection_info` dict containing info necessary to connect Jupyter clients to the scheduler.

**See also:**

[`Client.start\_ipython\_workers`](#) Start IPython on the workers

## Examples

```
>>> c.start_ipython_scheduler()
>>> %scheduler scheduler.processing
{'127.0.0.1:3595': {'inc-1', 'inc-2'},
 '127.0.0.1:53589': {'inc-2', 'add-5'}}
```

```
>>> c.start_ipython_scheduler(qtconsole=True)
```

**start\_ipython\_workers** (*workers=None, magic\_names=False, qtconsole=False, qtconsole\_args=None*)

Start IPython kernels on workers

**Parameters** **workers: list (optional)**

A list of worker addresses, defaults to all

**magic\_names: str or list(str) (optional)**

If defined, register IPython magics with these names for executing code on the workers. If string has asterix then expand asterix into 0, 1, ..., n for n workers

**qtconsole: bool (optional)**

If True, launch a Jupyter QtConsole connected to the worker(s).

**qtconsole\_args: list(str) (optional)**

Additional arguments to pass to the qtconsole on startup.

**Returns** `iter_connection_info`: list

List of `connection_info` dicts containing info necessary to connect Jupyter clients to the workers.

See also:

*Client.start\_ipython\_scheduler* start ipython on the scheduler

## Examples

```
>>> info = c.start_ipython_workers()
>>> %remote info['192.168.1.101:5752'] worker.data
{'x': 1, 'y': 100}
```

```
>>> c.start_ipython_workers('192.168.1.101:5752', magic_names='w')
>>> %w worker.data
{'x': 1, 'y': 100}
```

```
>>> c.start_ipython_workers('192.168.1.101:5752', qtconsole=True)
```

Add asterisk \* in magic names to add one magic per worker

```
>>> c.start_ipython_workers(magic_names='w_*')
>>> %w_0 worker.data
{'x': 1, 'y': 100}
>>> %w_1 worker.data
{'z': 5}
```

**submit** (*func*, \**args*, \*\**kwargs*)

Submit a function application to the scheduler

**Parameters** *func*: callable

**\*args:**

**\*\*kwargs:**

**pure: bool (defaults to True)**

Whether or not the function is pure. Set `pure=False` for impure functions like `np.random.random`.

**workers: set, iterable of sets**

A set of worker hostnames on which computations may be performed. Leave empty to default to all workers (common case)

**key: str**

Unique identifier for the task. Defaults to function-name and hash

**allow\_other\_workers: bool (defaults to False)**

Used with *workers*. Indicates whether or not the computations may be performed on workers that are not in the *workers* set(s).

**retries: int (default to 0)**

Number of allowed automatic retries if the task fails

**Returns** Future

See also:

*Client.map* Submit on many arguments at once

## Examples

```
>>> c = client.submit(add, a, b)
```

**unpublish\_dataset** (*name*, *\*\*kwargs*)  
Remove named datasets from scheduler

**See also:**

*Client.publish\_dataset*

## Examples

```
>>> c.list_datasets()
['my_dataset']
>>> c.unpublish_datasets('my_dataset')
>>> c.list_datasets()
[]
```

**upload\_file** (*filename*, *\*\*kwargs*)  
Upload local package to workers

This sends a local file up to all worker nodes. This file is placed into a temporary directory on Python's system path so any .py, .pyc, .egg or .zip files will be importable.

**Parameters filename: string**

Filename of .py, .pyc, .egg or .zip file to send to workers

## Examples

```
>>> client.upload_file('mylibrary.egg')
>>> from mylibrary import myfunc
>>> L = c.map(myfunc, seq)
```

**who\_has** (*futures=None*, *\*\*kwargs*)  
The workers storing each future's data

**Parameters futures: list (optional)**

A list of futures, defaults to all data

**See also:**

*Client.has\_what*, *Client.ncores*

## Examples

```
>>> x, y, z = c.map(inc, [1, 2, 3])
>>> wait([x, y, z])
>>> c.who_has()
{'inc-1c8dd6be1c21646c71f76c16d09304ea': ['192.168.1.141:46784'],
 'inc-1e297fc27658d7b67b3a758f16bcf47a': ['192.168.1.141:46784'],
 'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b': ['192.168.1.141:46784']}
```

```
>>> c.who_has([x, y])
{'inc-1c8dd6be1c21646c71f76c16d09304ea': ['192.168.1.141:46784'],
 'inc-1e297fc27658d7b67b3a758f16bcf47a': ['192.168.1.141:46784']}
```

**class** `distributed.Future` (*key*, *client=None*, *inform=True*, *state=None*)

A remotely running computation

A Future is a local proxy to a result running on a remote worker. A user manages future objects in the local Python process to determine what happens in the larger cluster.

**Parameters** **key:** `str`, or `tuple`

Key of remote data to which this future refers

**client:** `Client`

Client that should own this future. Defaults to `_get_global_client()`

**inform:** `bool`

Do we inform the scheduler that we need an update on this future

**See also:**

`Client` Creates futures

## Examples

Futures typically emerge from Client computations

```
>>> my_future = client.submit(add, 1, 2)
```

We can track the progress and results of a future

```
>>> my_future
<Future: status: finished, key: add-8f6e709446674bad78ea8aeecfee188e>
```

We can get the result or the exception and traceback from the future

```
>>> my_future.result()
```

**add\_done\_callback** (*fn*)

Call callback on future when callback has finished

The callback `fn` should take the future as its only argument. This will be called regardless of if the future completes successfully, errs, or is cancelled

The callback is executed in a separate thread.

**cancel** (*\*\*kwargs*)

Cancel request to run this future

**See also:**

`Client.cancel`

**cancelled** ()

Returns True if the future has been cancelled

**done** ()

Is the computation complete?

**exception** (*timeout=None, \*\*kwargs*)

Return the exception of a failed task

If *timeout* seconds are elapsed before returning, a `TimeoutError` is raised.

**See also:**

`Future.traceback`

**result** (*timeout=None*)

Wait until computation completes, gather result to local process.

If *timeout* seconds are elapsed before returning, a `TimeoutError` is raised.

**traceback** (*timeout=None, \*\*kwargs*)

Return the traceback of a failed task

This returns a traceback object. You can inspect this object using the `traceback` module. Alternatively if you call `future.result()` this traceback will accompany the raised exception.

If *timeout* seconds are elapsed before returning, a `TimeoutError` is raised.

**See also:**

`Future.exception`

## Examples

```
>>> import traceback
>>> tb = future.traceback()
>>> traceback.export_tb(tb)
[...]
```

**class** `distributed.Queue` (*name=None, client=None, maxsize=0*)

Distributed Queue

This allows multiple clients to share futures or small bits of data between each other with a multi-producer/multi-consumer queue. All metadata is sequentialized through the scheduler.

Elements of the Queue must be either Futures or msgpack-encodable data (ints, strings, lists, dicts). All data is sent through the scheduler so it is wise not to send large objects. To share large objects scatter the data and share the future instead.

**Warning:** This object is experimental and has known issues in Python 2

**See also:**

**Variable** shared variable between clients

## Examples

```
>>> from dask.distributed import Client, Queue
>>> client = Client()
>>> queue = Queue('x')
>>> future = client.submit(f, x)
>>> queue.put(future)
```

**get** (*timeout=None, batch=False, \*\*kwargs*)  
Get data from the queue

**Parameters** **timeout: Number (optional)**

Time in seconds to wait before timing out

**batch: boolean, int (optional)**

If True then return all elements currently waiting in the queue. If an integer than return that many elements from the queue If False (default) then return one item at a time

**put** (*value, timeout=None, \*\*kwargs*)  
Put data into the queue

**qsize** (*\*\*kwargs*)  
Current number of elements in the queue

**class** `distributed.Variable` (*name=None, client=None, maxsize=0*)  
Distributed Global Variable

This allows multiple clients to share futures and data between each other with a single mutable variable. All metadata is sequentialized through the scheduler. Race conditions can occur.

Values must be either Futures or msgpack-encodable data (ints, lists, strings, etc..) All data will be kept and sent through the scheduler, so it is wise not to send too much. If you want to share a large amount of data then `scatter` it and share the future instead.

**Warning:** This object is experimental and has known issues in Python 2

See also:

[Queue](#) shared multi-producer/multi-consumer queue between clients

## Examples

```
>>> from dask.distributed import Client, Variable
>>> client = Client()
>>> x = Variable('x')
>>> x.set(123) # doctest: +SKIP
>>> x.get() # doctest: +SKIP
123
>>> future = client.submit(f, x)
>>> x.set(future)
```

**delete** ()  
Delete this variable

Caution, this affects all clients currently pointing to this variable.

**get** (*timeout=None, \*\*kwargs*)  
Get the value of this variable

**set** (*value, \*\*kwargs*)  
Set the value of this variable

**Parameters** **value: Future or object**

Must be either a Future or a msgpack-encodable value

## 4.11 Machine Learning

Dask facilitates machine learning, statistics, and optimization workloads in a variety of ways. Generally Dask tries to support other high-quality solutions within the PyData ecosystem rather than reinvent new systems. Dask makes it easier to scale single-machine libraries like Scikit-Learn where possible and makes using distributed libraries like XGBoost or Tensorflow more comfortable for everyday users.

See the separate [Dask-ML documentation](#) for more information.

### 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.

- [Distributed Scheduling](#)
- [Scheduler Overview](#)
- [Single machine scheduler](#)
- [Scheduling in Depth](#)

## 4.12 Distributed Scheduling

Dask can run on a cluster of hundreds of machines and thousands of cores. Technical documentation for the distributed system is located on a separate website located here:

- <https://distributed.readthedocs.io/en/latest/>

## 4.13 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.get`: a synchronous scheduler, good for debugging
- **`distributed.Client.get`: a distributed scheduler for executing graphs** on multiple machines. This lives in the external [distributed](#) project.

### 4.13.1 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:

```
>>> from operator import add
>>> 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]
```

### 4.13.2 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*.

### 4.13.3 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
```

### 4.13.4 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 `dask.set_options`. This can be used either as a context manager, or to set the scheduler globally:

```
# As a context manager
>>> with dask.set_options(get=dask.multiprocessing.get):
...     x.sum().compute()

# Set globally
>>> dask.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 `dask.set_options`:

```
>>> from multiprocessing.pool import ThreadPool
>>> with dask.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.

### 4.13.5 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.get`. This runs everything serially, allowing it to work well with `pdb`:

```
>>> dask.set_options(get=dask.get)
>>> 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](#).

### 4.13.6 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

## 4.14 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.local.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.get` respectively.

### 4.14.1 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](#).

### 4.14.2 Performance

*EDIT: The experiments run in this section are now outdated. Anecdotal testing shows that performance has improved significantly. There is now about 200 us overhead per task and about 1 ms startup time.*

**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
```

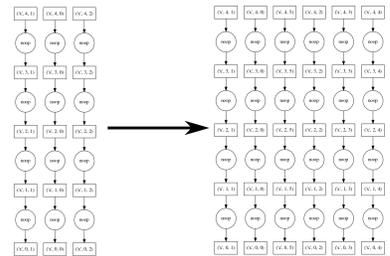
Most of this overhead is from spinning up a ThreadPool each time. This may be mediated by using a global or contextual pool:

```
>>> from multiprocessing.pool import ThreadPool
>>> pool = ThreadPool()
>>> da.set_options(pool=pool) # set global threadpool

or

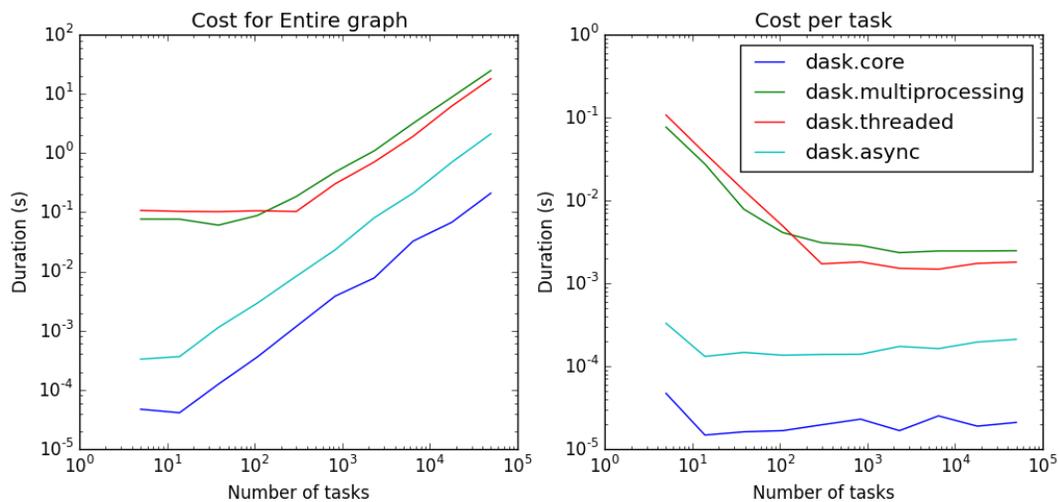
>>> with set_options(pool=pool) # use threadpool throughout with block
...     ...
```

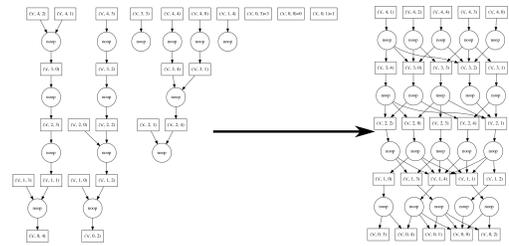
We now measure scaling the number of tasks and scaling the density of the graph:



### 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.

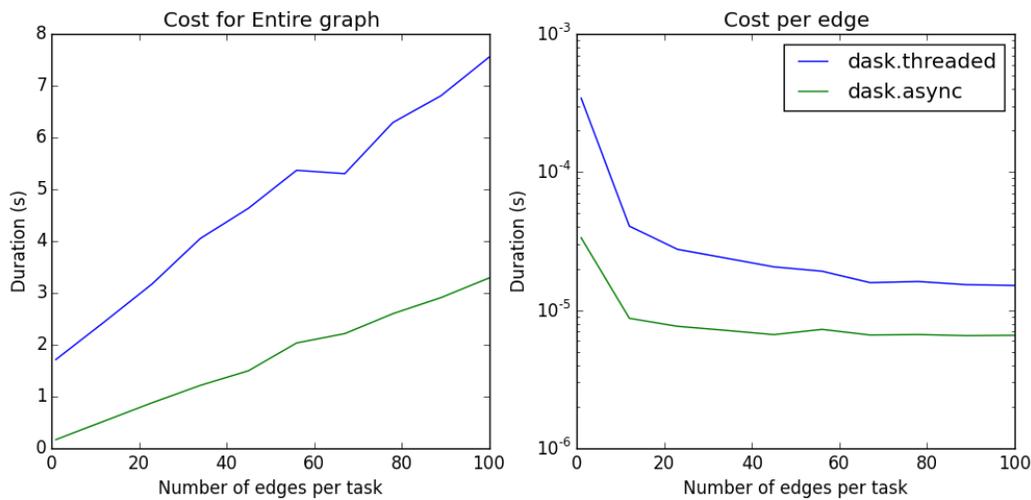




## Linear scaling with number of edges

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](#)

## 4.14.3 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.

## 4.15 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/scheduler.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*

## 4.16 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

### 4.16.1 `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))
>>> dict(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

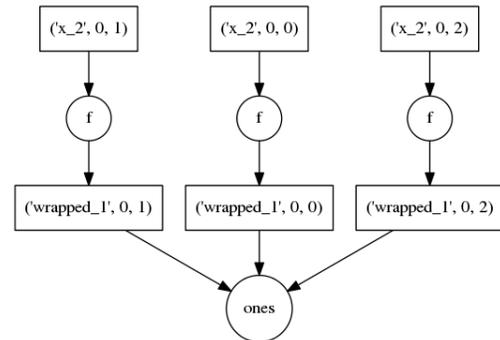
```
>>> dict((x + 1).dask)
{'wrapped_1', 0, 0): (ones, (5, 5)),
 ('wrapped_1', 0, 1): (ones, (5, 5)),
 ('wrapped_1', 0, 2): (ones, (5, 5))
 ('x_1', 0, 0): (add, ('wrapped_1', 0, 0), 1),
 ('x_1', 0, 1): (add, ('wrapped_1', 0, 1), 1),
 ('x_1', 0, 2): (add, ('wrapped_1', 0, 2), 1)}
```

---

**Note:** In this example we use simple names like `x_1`, `ones`, and `add` for demonstration purposes. However in practice these names may be more complex and include long hashed names.

---

## 4.16.2 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.

## 4.17 Diagnostics

Profiling parallel code can be tricky, but `dask.diagnostics` provides functionality to aid in profiling and inspecting `dask` graph execution.

### 4.17.1 Scheduler Callbacks

Schedulers based on `dask.local.get_async` (currently `dask.get`, `dask.threaded.get`, and `dask.multiprocessing.get`) accept five 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 5, stored in the order presented above. Callbacks for common use cases are provided in `dask.diagnostics`.

### 4.17.2 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()
```

### 4.17.3 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() 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 350%.

#### 4.17.4 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'!
```

## Graphs

Internally Dask encodes 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 rare unless you intend to develop new modules with Dask. Even then, *dask.delayed* is often a better choice. If you are a *core developer*, then you should start here.

- [Overview](#)
- [Specification](#)
- [Custom Graphs](#)
- [Optimization](#)

## 4.18 Overview

An explanation of dask task graphs.

### 4.18.1 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.

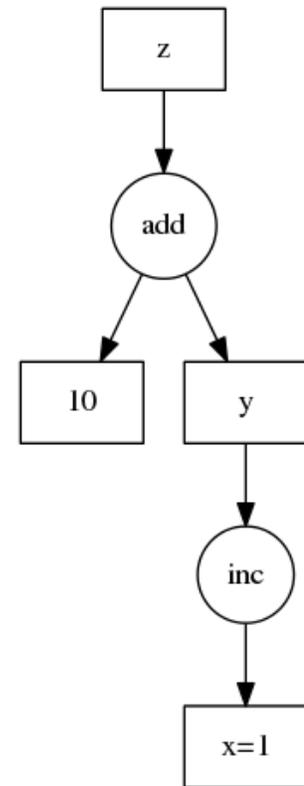
Commonly a desire for parallel execution causes this shift of responsibility from compiler to human developer. In these cases, we often represent the structure of our program explicitly as data within the program itself.

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.

### 4.18.2 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.

### 4.18.3 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.

## 4.19 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.

### 4.19.1 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`.

## 4.19.2 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.

## 4.19.3 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.

## 4.19.4 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.

## 4.20 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.

### 4.20.1 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:

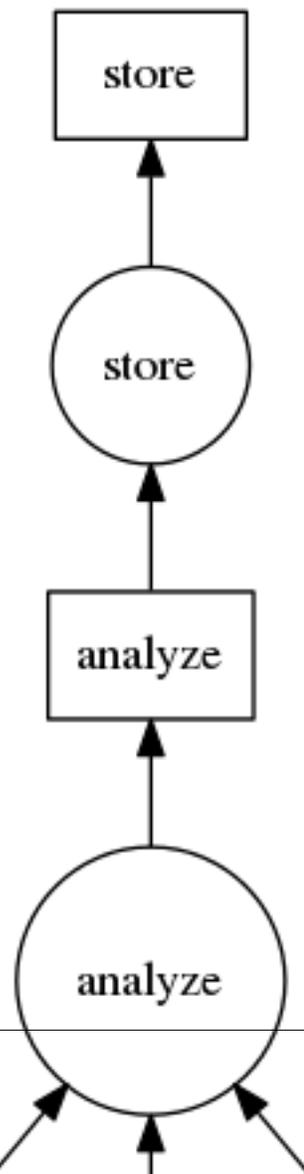
```
def load(filename):
    ...

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
```

## 4.20.2 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.

## 4.21 Optimization

Performance can be significantly improved in different contexts by making

small optimizations on the dask graph before calling the scheduler.

The `dask.optimization` 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.

### 4.21.1 Example

Suppose you had a custom dask graph for doing a word counting task:

```
>>> from __future__ import _
↳ print_function

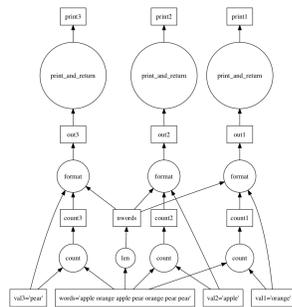
>>> def print_and_
↳ return(string):
...     print(string)
...     return string

>>> 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 occurrence 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

>>> outputs = ['print1', 'print2']
>>> results = get(dsk, outputs)
word list has 2 occurrences of apple, out of 7 words
word list has 2 occurrences of orange, out of 7 words

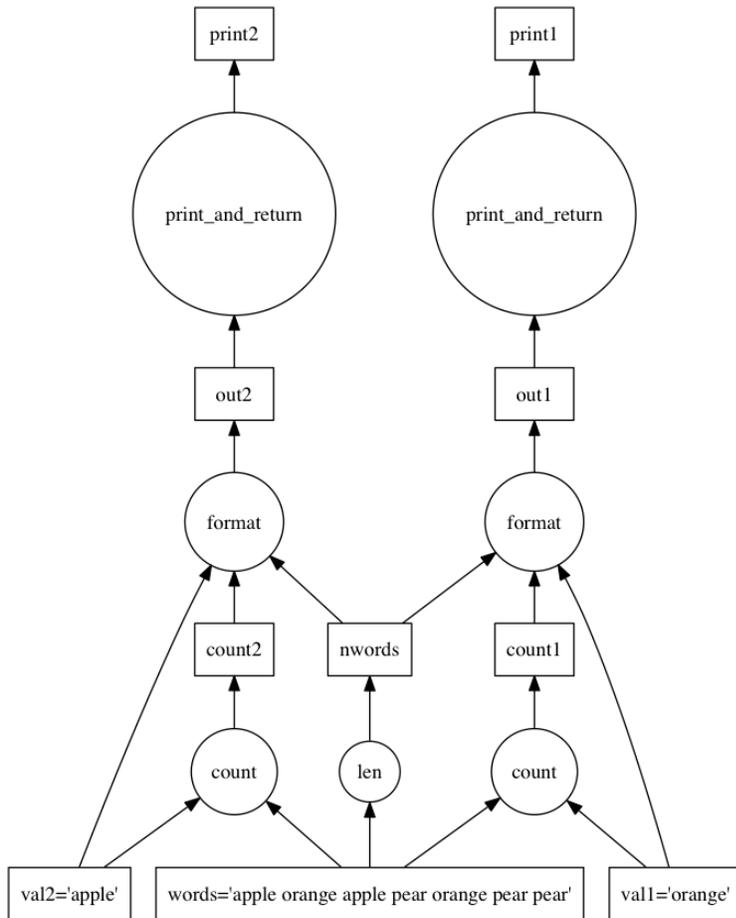
>>> results
('word list has 2 occurrences of orange, out of 7 words',
 'word list has 2 occurrences of apple, out of 7 words')

```

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

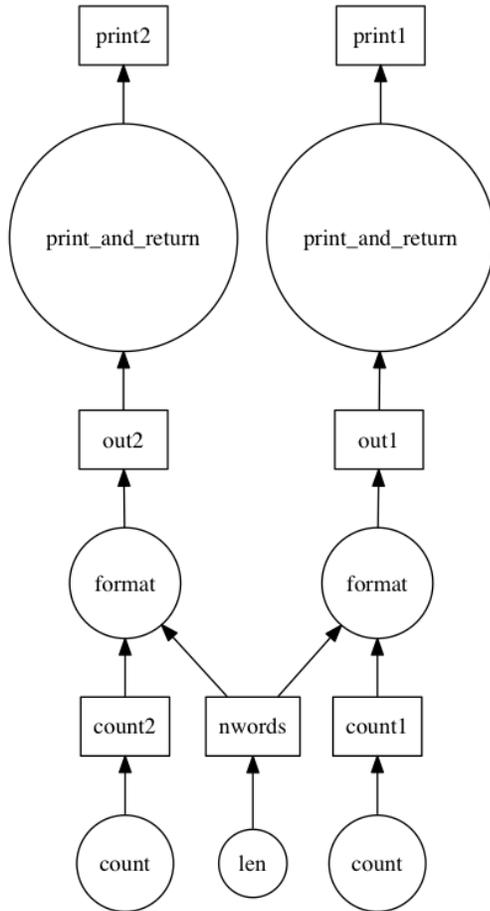
amount of work done in later steps:

```
>>> from dask.optimization import cull
>>> dsk1, dependencies = cull(dsk, outputs)
```



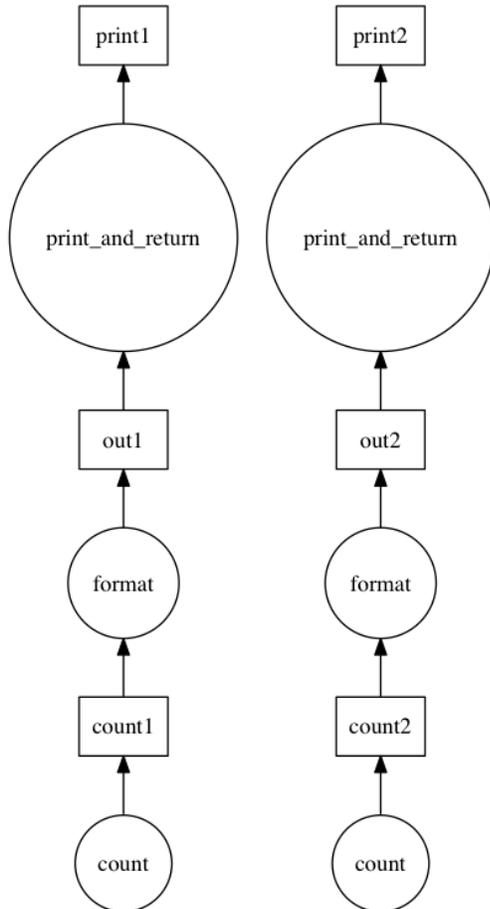
Looking at the task graph above, there are multiple accesses to constants such as 'val1' or 'val2' in the dask graph. These can be inlined into the tasks to improve efficiency using the `inline` function. For example:

```
>>> from dask.optimization import inline
>>> dsk2 = inline(dsk1, dependencies=dependencies)
>>> results = get(dsk2, outputs)
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:

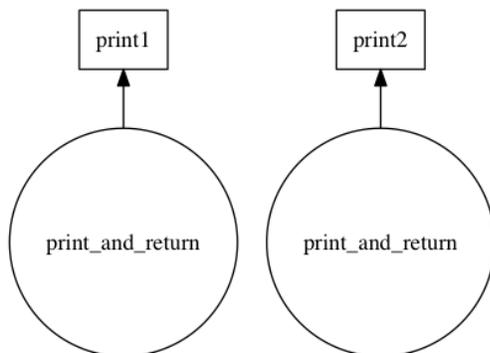
```
>>> from dask.optimization import inline_functions
>>> dsk3 = inline_functions(dsk2, outputs, [len, str.split],
...                        dependencies=dependencies)
>>> results = get(dsk3, outputs)
word list has 2 occurrences of apple, out of 7 words
word list has 2 occurrences of orange, out of 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.optimization import fuse
>>> dsk4, dependencies = fuse(dsk3)
>>> results = get(dsk4, outputs)
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, keys, [len, str.split],
...                               dependencies=deps)
...     dsk4, deps = fuse(dsk3)
...     return get(dsk4, keys)

>>> optimize_and_get(dsk, outputs)
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 unnecessary 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.optimization`. For more information, see the API below.

### 4.21.2 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.  $a + a \rightarrow 2*a$
2.  $a * a \rightarrow a**2$

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.

### 4.21.3 Keyword Arguments

Some optimizations take optional keyword arguments. To pass keywords from the compute call down to the right optimization, prepend the keyword with the name of the optimization. For example to send a `keys=` keyword argument to the `fuse` optimization from a compute call, use the `fuse_keys=` keyword:

```
def fuse(dsk, keys=None):
    ...

x.compute(fuse_keys=['x', 'y', 'z'])
```

### 4.21.4 Customizing Optimization

Dask defines a default optimization strategy for each collection type (Array, Bag, DataFrame, Delayed). However different applications may have different needs. To address this variability of needs, you can construct your own custom optimization function and use it instead of the default. An optimization function takes in a task graph and list of desired keys and returns a new task graph.

```
def my_optimize_function(dsk, keys):
    new_dsk = {...}
    return new_dsk
```

You can then register this optimization class against whichever collection type you prefer and it will be used instead of the default scheme.

```
with dask.set_options(array_optimize=my_optimize_function):
    x, y = dask.compute(x, y)
```

You can register separate optimization functions for different collections, or you can register `None` if you do not want particular types of collections to be optimized.

```
with dask.set_options(array_optimize=my_optimize_function,
                      dataframe_optimize=None,
                      delayed_optimize=my_other_optimize_function):
    ...
```

You need not specify all collections. Collections will default to their standard optimization scheme (which is usually a good choice).

## 4.21.5 API

### Top level optimizations

---

cull

---

fuse

---

inline

---

inline\_functions

---

### Utility functions

---

functions\_of

---

### Rewrite Rules

---

<code>RewriteRule(lhs, rhs[, vars])</code>	A rewrite rule.
<code>RuleSet(*rules)</code>	A set of rewrite rules.

---

### Definitions

`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 <i>RewriteRule</i> 's included in the <i>RuleSet</i> .
-------	---

## Help & reference

- [Debugging](#)
- [Changelog](#)
- [Dask Cheat Sheet](#)
- [Presentations On Dask](#)
- [Development Guidelines](#)
- [Frequently Asked Questions](#)
- [Comparison to Spark](#)
- [Opportunistic Caching](#)
- [Internal Data Ingestion](#)
- [Remote Data Services](#)
- [Custom Collections](#)
- [Citations](#)

## 4.22 Debugging

Debugging parallel programs is hard. Normal debugging tools like logging and using `pdb` to interact with tracebacks stop working normally when exceptions occur in far-away machines or different processes or threads.

Dask has a variety of mechanisms to make this process easier. Depending on your situation some of these approaches may be more appropriate than others.

These approaches are ordered from lightweight or easy solutions to more involved solutions.

### 4.22.1 Exceptions

When a task in your computation fails the standard way of understanding what went wrong is to look at the exception and traceback. Often people do this with the `pdb` module, IPython `%debug` or `%pdb` magics, or by just looking at the traceback and investigating where in their code the exception occurred.

Normally when a computation computes in a separate thread or a different machine these approaches break down. Dask provides a few mechanisms to recreate the normal Python debugging experience.

## Inspect Exceptions and Tracebacks

By default, Dask already copies the exception and traceback wherever they occur and reraises that exception locally. If your task failed with a `ZeroDivisionError` remotely then you'll get a `ZeroDivisionError` in your interactive session. Similarly you'll see a full traceback of where this error occurred, which, just like in normal Python, can help you to identify the troublesome spot in your code.

However, you cannot use the `pdb` module or `%debug` IPython magics with these tracebacks to look at the value of variables during failure. You can only inspect things visually. Additionally, the top of the traceback may be filled with functions that are dask-specific and not relevant to your problem, you can safely ignore these.

Both the single-machine and distributed schedulers do this.

## Use the Single-Threaded Scheduler

Dask ships with a simple single-threaded scheduler. This doesn't offer any parallel performance improvements, but does run your Dask computation faithfully in your local thread, allowing you to use normal tools like `pdb`, `%debug` IPython magics, the profiling tools like the `cProfile` module and `snakeviz`. This allows you to use all of your normal Python debugging tricks in Dask computations, as long as you don't need parallelism.

This only works for single-machine schedulers. It does not work with `dask.distributed` unless you are comfortable using the Tornado API (look at the [testing infrastructure](#) docs, which accomplish this). Also, because this operates on a single machine it assumes that your computation can run on a single machine without exceeding memory limits. It may be wise to use this approach on smaller versions of your problem if possible.

## Rerun Failed Task Locally

If a remote task fails, we can collect the function and all inputs, bring them to the local thread, and then rerun the function in hopes of triggering the same exception locally, where normal debugging tools can be used.

With the single machine schedulers, use the `rerun_exceptions_locally=True` keyword.

```
x.compute(rerun_exceptions_locally=True)
```

On the distributed scheduler use the `recreate_error_locally` method on anything that contains Futures :

```
>>> x.compute()
ZeroDivisionError(...)

>>> %pdb
>>> future = client.compute(x)
>>> client.recreate_error_locally(future)
```

## Remove Failed Futures Manually

Sometimes only parts of your computations fail, for example if some rows of a CSV dataset are faulty in some way. When running with the distributed scheduler you can remove chunks of your data that have produced bad results if you switch to dealing with Futures.

```
>>> import dask.dataframe as dd
>>> df = ...           # create dataframe
>>> df = df.persist() # start computing on the cluster

>>> from distributed.client import futures_of
>>> futures = futures_of(df) # get futures behind dataframe
>>> futures
[<Future: status: finished, type: pd.DataFrame, key: load-1>
 <Future: status: finished, type: pd.DataFrame, key: load-2>
 <Future: status: error, key: load-3>
 <Future: status: pending, key: load-4>
 <Future: status: error, key: load-5>]

>>> # wait until computation is done
>>> while any(f.status == 'pending' for f in futures):
...     sleep(0.1)

>>> # pick out only the successful futures and reconstruct the dataframe
>>> good_futures = [f for f in futures if f.status == 'finished']
>>> df = dd.from_delayed(good_futures, meta=df._meta)
```

This is a bit of a hack, but often practical when first exploring messy data. If you are using the `concurrent.futures` API (`map`, `submit`, `gather`) then this approach is more natural.

## 4.22.2 Inspect Scheduling State

Not all errors present themselves as Exceptions. For example in a distributed system workers may die unexpectedly or your computation may be unreasonably slow due to inter-worker communication or scheduler overhead or one of several other issues. Getting feedback about what's going on can help to identify both failures and general performance bottlenecks.

For the single-machine scheduler see *diagnostics* documentation. The rest of the section will assume that you are using the `distributed` scheduler where these issues arise more commonly.

### Web Diagnostics

First, the distributed scheduler has a number of [diagnostic web pages](#) showing dozens of recorded metrics like CPU, memory, network, and disk use, a history of previous tasks, allocation of tasks to workers, worker memory pressure, work stealing, open file handle limits, etc.. *Many* problems can be correctly diagnosed by inspecting these pages. By default these are available at `http://scheduler:8787/` `http://scheduler:8788/` and `http://worker:8789/` where `scheduler` and `worker` should be replaced by the addresses of the scheduler and each of the workers. See [web diagnostic docs](#) for more information.

### Logs

The scheduler and workers and client all emits logs using Python's standard [logging module](#). By default these emit to standard error. When Dask is launched by a cluster job scheduler (SGE/SLURM/YARN/Mesos/Marathon/Kubernetes/whatever) that system will track these logs and will have an interface to help you access them. If you are launching Dask on your own they will probably dump to the screen unless you [redirect stderr to a file](#) .

You can control the logging verbosity in the `~/dask/config.yaml` file. Defaults currently look like the following:

```
logging:
  distributed: info
  distributed.client: warning
  bokeh: error
```

So for example you could add a line like `distributed.worker: debug` to get *very* verbose output from the workers.

### 4.22.3 LocalCluster

If you are using the distributed scheduler from a single machine you may be setting up workers manually using the command line interface or you may be using `LocalCluster` which is what runs when you just call `Client()`

```
>>> from dask.distributed import Client, LocalCluster
>>> client = Client() # This is actually the following two commands

>>> cluster = LocalCluster()
>>> client = Client(cluster.scheduler.address)
```

`LocalCluster` is useful because the scheduler and workers are in the same process with you, so you can easily inspect their *state* while they run (they are running in a separate thread).

```
>>> cluster.scheduler.processing
{'worker-one:59858': {'inc-123', 'add-443'},
 'worker-two:48248': {'inc-456'}}
```

You can also do this for the workers *if* you run them without nanny processes.

```
>>> cluster = LocalCluster(nanny=False)
>>> client = Client(cluster)
```

This can be very helpful if you want to use the `dask.distributed` API and still want to investigate what is going on directly within the workers. Information is not distilled for you like it is in the web diagnostics, but you have full low-level access.

### 4.22.4 Inspect state with IPython

Sometimes you want to inspect the state of your cluster, but you don't have the luxury of operating on a single machine. In these cases you can launch an IPython kernel on the scheduler and on every worker, which lets you inspect state on the scheduler and workers as computations are completing.

This does not give you the ability to run `%pdb` or `%debug` on remote machines, the tasks are still running in separate threads, and so are not easily accessible from an interactive IPython session.

For more details, see the [Dask.distributed IPython docs](#).

## 4.23 Changelog

### 4.23.1 0.16.2 / 2018-MM-DD

#### Array

- Update error handling when `len` is called with empty chunks ([GH#3058](#)) Xander Johnson

- Fixes a metadata bug with store's `return_stored` option (GH#3064) John A Kirkham
- Fix a bug in `optimization.fuse_slice` to properly handle when first input is `None` (GH#3076) James Bourbeau

### DataFrame

### Bag

### Core

- Change default task ordering to prefer nodes with few dependents and then many downstream dependencies (GH#3056) Matthew Rocklin
- Add `color=` option to visualize to color by task order (GH#3057) Matthew Rocklin
- Deprecate `dask.bytes.open_text_files` (GH#3077) Jim Crist
- Remove short-circuit hdfs reads handling due to maintenance costs. May be re-added in a more robust manner later (GH#3079) Jim Crist
- Add `dask.base.optimize` for optimizing multiple collections without computing. (GH#3071) Jim Crist
- Rename `dask.optimize` module to `dask.optimization` (GH#3071) Jim Crist

## 4.23.2 0.16.1 / 2018-01-09

### Array

- Fix handling of scalar percentile values in `percentile` (GH#3021) James Bourbeau
- Prevent `bool()` coercion from calling `compute` (GH#2958) Albert DeFusco
- Add `matmul` (GH#2904) John A Kirkham
- Support N-D arrays with `matmul` (GH#2909) John A Kirkham
- Add `vdot` (GH#2910) John A Kirkham
- Explicit `chunks` argument for `broadcast_to` (GH#2943) Stephan Hoyer
- Add `meshgrid` (GH#2938) John A Kirkham and (GH#3001) Markus Gonser
- Preserve singleton chunks in `fftshift/ifftshift` (GH#2733) John A Kirkham
- Fix handling of negative indexes in `vindex` and raise errors for out of

bounds indexes (GH#2967) Stephan Hoyer - Add `flip`, `flipud`, `fliplr` (GH#2954) John A Kirkham - Add `float_power` `ufunc` (GH#2962) (GH#2969) John A Kirkham - Compatibility for changes to structured arrays in the upcoming NumPy 1.14 release (GH#2964) Tom Augspurger - Add `block` (GH#2650) John A Kirkham - Add `frompyfunc` (GH#3030) Jim Crist - Add the `return_stored` option to `store` for chaining stored results (GH#2980) John A Kirkham

### DataFrame

- Fixed naming bug in cumulative aggregations (GH#3037) Martijn Arts
- Fixed `dd.read_csv` when `names` is given but `header` is not set to `None` (GH#2976) Martijn Arts

- Fixed `dd.read_csv` so that passing instances of `CategoricalDtype` in `dtype` will result in known categoricals (GH#2997) Tom Augspurger
- Prevent `bool()` coercion from calling `compute` (GH#2958) Albert DeFusco
- `DataFrame.read_sql()` (GH#2928) to an empty database tables returns an empty dask dataframe ‘Apostolos Vlachopoulos’\_
- Compatibility for reading Parquet files written by PyArrow 0.8.0 (GH#2973) Tom Augspurger
- Correctly handle the column name (`df.columns.name`) when reading in `dd.read_parquet (:pr:2973)` Tom Augspurger
- Fixed `dd.concat` losing the index dtype when the data contained a categorical (GH#2932) Tom Augspurger
- Add `dd.Series.rename` (GH#3027) Jim Crist
- `DataFrame.merge()` now supports merging on a combination of columns and the index (GH#2960) Jon Mease
- Removed the deprecated `dd.rolling*` methods, in preparation for their removal in the next pandas release (GH#2995) Tom Augspurger
- Fix metadata inference bug in which single-partition series were mistakenly special cased (GH#3035) Jim Crist
- Add support for `Series.str.cat` (GH#3028) Jim Crist

## Core

- Improve 32-bit compatibility (GH#2937) Matthew Rocklin
- Change task prioritization to avoid upwards branching (GH#3017) Matthew Rocklin

### 4.23.3 0.16.0 / 2017-11-17

This is a major release. It includes breaking changes, new protocols, and a large number of bug fixes.

## Array

- Add `atleast_1d`, `atleast_2d`, and `atleast_3d` (GH#2760) (GH#2765) John A Kirkham
- Add `allclose` (GH#2771) by John A Kirkham
- Remove `random.different_seeds` from Dask Array API docs (GH#2772) John A Kirkham
- Deprecate `vnorm` in favor of `dask.array.linalg.norm` (GH#2773) John A Kirkham
- Reimplement `unique` to be lazy (GH#2775) John A Kirkham
- Support broadcasting of Dask Arrays with 0-length dimensions (GH#2784) John A Kirkham
- Add `asarray` and `asanyarray` to Dask Array API docs (GH#2787) James Bourbeau
- Support `unique`’s `return_*` arguments (GH#2779) John A Kirkham
- Simplify `_unique_internal` (GH#2850) (GH#2855) John A Kirkham
- Avoid removing some getter calls in array optimizations (GH#2826) Jim Crist

## DataFrame

- Support `pyarrow` in `dd.to_parquet` (GH#2868) Jim Crist
- Fixed `DataFrame.quantile` and `Series.quantile` returning `nan` when missing values are present (GH#2791:) Tom Augspurger
- Fixed `DataFrame.quantile` losing the result `.name` when `q` is a scalar (GH#2791:) Tom Augspurger
- Fixed `dd.concat` return a `dask.DataFrame` when concatenating a single series along the columns, matching `pandas`' behavior (GH#2800) James Munroe
- Fixed default `inplace` parameter for `DataFrame.eval` to match the `pandas` default for `pandas >= 0.21.0` (GH#2838) Tom Augspurger
- Fix exception when calling `DataFrame.set_index` on text column where one of the partitions was empty (GH#2831) Jesse Vogt
- Do not raise exception when calling `DataFrame.set_index` on empty dataframe (GH#2827) 'Jess Vogt'
- Fixed bug in `Dataframe.fillna` when filling with a `Series` value (GH#2810) Tom Augspurger
- Deprecate old argument ordering in `dd.to_parquet` to better match convention of putting the dataframe first (GH#2867) Jim Crist
- `df.astype(categorical_dtype -> known_categoricals)` (GH#2835) Jim Crist
- Test against `Pandas` release candidate (GH#2814) Tom Augspurger
- Add more tests for `read_parquet(engine='pyarrow')` (GH#2822) Uwe Korn
- Remove unnecessary `map_partitions` in `aggregate` (GH#2712) Christopher Prohm
- Fix bug calling `sample` on empty partitions (GH#2818) @xwang777
- Error nicely when parsing dates in `read_csv` (GH#2863) Jim Crist
- Cleanup handling of passing filesystem objects to `PyArrow` readers (GH#2527) @fjetter
- Support repartitioning even if there are no divisions (GH#2873) @Ced4
- Support reading/writing to `hdfs` using `pyarrow` in `dd.to_parquet` (GH#2894:, GH#2881:) Jim Crist

## Core

- Allow tuples as `shardedict` keys (GH#2763) Matthew Rocklin
- Calling `compute` within a `dask.distributed` task defaults to `distributed` scheduler (GH#2762) Matthew Rocklin
- Auto-import `gcsfs` when `gcs://` protocol is used (GH#2776) Matthew Rocklin
- Fully remove `dask.async` module, use `dask.local` instead (GH#2828) Thomas Caswell
- Compatibility with `bokeh` 0.12.10 (GH#:2844) Tom Augspurger
- Reduce test memory usage (GH#2782) Jim Crist
- Add `Dask` collection interface (GH#2748) Jim Crist
- Update `Dask` collection interface during `XArray` integration (GH#2847) Matthew Rocklin
- Close resource profiler process on `__exit__` (GH#2871) Jim Crist
- Fix S3 tests (GH#2875) Jim Crist
- Fix port for `bokeh` dashboard in docs (GH#2889) Ian Hopkinson

- Wrap Dask filesystems for PyArrow compatibility (GH#2881) Jim Crist

#### 4.23.4 0.15.4 / 2017-10-06

##### Array

- `da.random.choice` now works with array arguments (GH#2781)
- Support indexing in arrays with `np.int` (fixes regression) (GH#2719)
- Handle zero dimension with rechunking (GH#2747)
- Support `-1` as an alias for “size of the dimension” in `chunks` (GH#2749)
- Call `mkdir` in `array.to_numpy_stack` (GH#2709)

##### DataFrame

- Added the `.str` accessor to Categoricals with string categories (GH#2743)
- Support `int96` (spark) datetimes in parquet writer (GH#2711)
- Pass on file scheme to `fastparquet` (GH#2714)
- Support Pandas 0.21 (GH#2737)

##### Bag

- Add tree reduction support for `foldby` (:pr: 2710)

##### Core

- Drop `s3fs` from `pip install dask[complete]` (GH#2750)

#### 4.23.5 0.15.3 / 2017-09-24

##### Array

- Add masked arrays (GH#2301)
- Add `*_like` array creation functions (GH#2640)
- Indexing with unsigned integer array (GH#2647)
- Improved slicing with boolean arrays of different dimensions (GH#2658)
- Support literals in `top` and `atop` (GH#2661)
- Optional axis argument in cumulative functions (GH#2664)
- Improve tests on scalars with `assert_eq` (GH#2681)
- Fix `norm` `keepdims` (GH#2683)
- Add `ptp` (GH#2691)
- Add `apply_along_axis` (GH#2690) and `apply_over_axes` (GH#2702)

## DataFrame

- Added `Series.str[index]` (GH#2634)
- Allow the `groupby` by param to handle columns and index levels (GH#2636)
- **DataFrame.to\_csv** and **Bag.to\_textfiles** now return the filenames to which they have written (GH#2655)
- Fix combination of `partition_on` and `append` in `to_parquet` (GH#2645)
- Fix for parquet file schemes (GH#2667)
- Repartition works with mixed categoricals (GH#2676)

## Core

- `python setup.py test` now runs tests (GH#2641)
- Added new cheatsheet (GH#2649)
- Remove resize tool in Bokeh plots (GH#2688)

## 4.23.6 0.15.2 / 2017-08-25

### Array

- Remove spurious keys from `map_overlap` graph (GH#2520)
- `where` works with non-bool condition and scalar values (GH#2543) (GH#2549)
- Improve `compress` (GH#2541) (GH#2545) (GH#2555)
- Add `argwhere`, `_nonzero`, and `where(cond)` (GH#2539)
- Generalize `vindex` in `dask.array` to handle multi-dimensional indices (GH#2573)
- Add `choose` method (GH#2584)
- Split code into reorganized files (GH#2595)
- Add `linalg.norm` (GH#2597)
- Add `diff`, `ediff1d` (GH#2607), (GH#2609)
- Improve `dtype` inference and reflection (GH#2571)

### Bag

- Remove deprecated Bag behaviors (GH#2525)

### DataFrame

- Support callables in `assign` (GH#2513)
- better error messages for `read_csv` (GH#2522)
- Add `dd.to_timedelta` (GH#2523)
- Verify metadata in `from_delayed` (GH#2534) (GH#2591)

- Add DataFrame.isin (GH#2558)
- Read\_hdf supports iterables of files (GH#2547)

## Core

- Remove bare `except :` blocks everywhere (GH#2590)

### 4.23.7 0.15.1 / 2017-07-08

- Add storage\_options to to\_textfiles and to\_csv (GH#2466)
- Rechunk and simplify rfftfreq (GH#2473), (GH#2475)
- Better support ndarray subclasses (GH#2486)
- Import star in dask.distributed (GH#2503)
- Threadsafe cache handling with tokenization (GH#2511)

### 4.23.8 0.15.0 / 2017-06-09

## Array

- Add dask.array.stats submodule (GH#2269)
- Support `ufunc.outer` (GH#2345)
- Optimize fancy indexing by reducing graph overhead (GH#2333) (GH#2394)
- Faster array tokenization using alternative hashes (GH#2377)
- Added the `matmul @` operator (GH#2349)
- Improved coverage of the `numpy.fft` module (GH#2320) (GH#2322) (GH#2327) (GH#2323)
- Support NumPy's `__array_ufunc__` protocol (GH#2438)

## Bag

- Fix bug where reductions on bags with no partitions would fail (GH#2324)
- Add broadcasting and variadic `db.map` top-level function. Also remove auto-expansion of tuples as map arguments (GH#2339)
- Rename `Bag.concat` to `Bag.flatten` (GH#2402)

## DataFrame

- Parquet improvements (GH#2277) (GH#2422)

## Core

- Move dask.async module to dask.local (GH#2318)
- Support callbacks with nested scheduler calls (GH#2397)
- Support pathlib.Path objects as uris (GH#2310)

### 4.23.9 0.14.3 / 2017-05-05

#### DataFrame

- Pandas 0.20.0 support

### 4.23.10 0.14.2 / 2017-05-03

#### Array

- Add da.indices (GH#2268), da.tile (GH#2153), da.roll (GH#2135)
- Simultaneously support drop\_axis and new\_axis in da.map\_blocks (GH#2264)
- Rechunk and concatenate work with unknown chunksizes (GH#2235) and (GH#2251)
- Support non-numpy container arrays, notably sparse arrays (GH#2234)
- Tensor dot contracts over multiple axes (GH#2186)
- Allow delayed targets in da.store (GH#2181)
- Support interactions against lists and tuples (GH#2148)
- Constructor plugins for debugging (GH#2142)
- Multi-dimensional FFTs (single chunk) (GH#2116)

#### Bag

- to\_dataframe enforces consistent types (GH#2199)

#### DataFrame

- Set\_index always fully sorts the index (GH#2290)
- Support compatibility with pandas 0.20.0 (GH#2249), (GH#2248), and (GH#2246)
- Support Arrow Parquet reader (GH#2223)
- Time-based rolling windows (GH#2198)
- Repartition can now create more partitions, not just less (GH#2168)

## Core

- Always use absolute paths when on POSIX file system (GH#2263)
- Support user provided graph optimizations (GH#2219)
- Refactor path handling (GH#2207)
- Improve fusion performance (GH#2129), (GH#2131), and (GH#2112)

### 4.23.11 0.14.1 / 2017-03-22

## Array

- Micro-optimize optimizations (GH#2058)
- Change slicing optimizations to avoid fusing raw numpy arrays (GH#2075) (GH#2080)
- Dask.array operations now work on numpy arrays (GH#2079)
- Reshape now works in a much broader set of cases (GH#2089)
- Support deepcopy python protocol (GH#2090)
- Allow user-provided FFT implementations in `da.fft` (GH#2093)

## Bag

### DataFrame

- Fix `to_parquet` with empty partitions (GH#2020)
- Optional `npartitions='auto'` mode in `set_index` (GH#2025)
- Optimize shuffle performance (GH#2032)
- Support efficient repartitioning along time windows like `repartition(freq='12h')` (GH#2059)
- Improve speed of `categorize` (GH#2010)
- Support single-row dataframe arithmetic (GH#2085)
- Automatically avoid shuffle when setting index with a sorted column (GH#2091)
- Improve handling of integer-na handling in `read_csv` (GH#2098)

## Delayed

- Repeated attribute access on delayed objects uses the same key (GH#2084)

## Core

- Improve naming of nodes in dot visuals to avoid generic `apply` (GH#2070)
- Ensure that worker processes have different random seeds (GH#2094)

## 4.23.12 0.14.0 / 2017-02-24

### Array

- Fix corner cases with zero shape and misaligned values in `arange` (GH#1902), (GH#1904), (GH#1935), (GH#1955), (GH#1956)
- Improve concatenation efficiency (GH#1923)
- Avoid hashing in `from_array` if name is provided (GH#1972)

### Bag

- Repartition can now increase number of partitions (GH#1934)
- Fix bugs in some reductions with empty partitions (GH#1939), (GH#1950), (GH#1953)

### DataFrame

- Support non-uniform categoricals (GH#1877), (GH#1930)
- Groupby cumulative reductions (GH#1909)
- `DataFrame.loc` indexing now supports lists (GH#1913)
- Improve multi-level groupbys (GH#1914)
- Improved HTML and string repr for DataFrames (GH#1637)
- Parquet append (GH#1940)
- Add `dd.demo.daily_stock` function for teaching (GH#1992)

### Delayed

- Add `traverse=` keyword to `delayed` to optionally avoid traversing nested data structures (GH#1899)
- Support Futures in `from_delayed` functions (GH#1961)
- Improve serialization of decorated delayed functions (GH#1969)

### Core

- Improve windows path parsing in corner cases (GH#1910)
- Rename tasks when fusing (GH#1919)
- Add top level `persist` function (GH#1927)
- Propagate `errors=` keyword in byte handling (GH#1954)
- `Dask.compute` traverses Python collections (GH#1975)
- Structural sharing between graphs in `dask.array` and `dask.delayed` (GH#1985)

### 4.23.13 0.13.0 / 2017-01-02

#### Array

- Mandatory dtypes on `dask.array`. All operations maintain dtype information and UDF functions like `map_blocks` now require a `dtype=` keyword if it can not be inferred. (GH#1755)
- Support arrays without known shapes, such as arises when slicing arrays with arrays or converting dataframes to arrays (GH#1838)
- Support mutation by setting one array with another (GH#1840)
- Tree reductions for covariance and correlations. (GH#1758)
- Add `SerializableLock` for better use with distributed scheduling (GH#1766)
- Improved `atop` support (GH#1800)
- Rechunk optimization (GH#1737), (GH#1827)

#### Bag

- Avoid wrong results when recomputing the same groupby twice (GH#1867)

#### DataFrame

- Add `map_overlap` for custom rolling operations (GH#1769)
- Add `shift` (GH#1773)
- Add Parquet support (GH#1782) (GH#1792) (GH#1810), (GH#1843), (GH#1859), (GH#1863)
- Add missing methods `combine`, `abs`, `autocorr`, `sem`, `nsmallest`, `first`, `last`, `prod`, (GH#1787)
- Approximate `nunique` (GH#1807), (GH#1824)
- Reductions with multiple output partitions (for operations like `drop_duplicates`) (GH#1808), (GH#1823) (GH#1828)
- Add `delitem` and `copy` to DataFrames, increasing mutation support (GH#1858)

#### Delayed

- Changed behaviour for `delayed(nout=0)` and `delayed(nout=1)`: `delayed(nout=1)` does not default to `out=None` anymore, and `delayed(nout=0)` is also enabled. I.e. functions with return tuples of length 1 or 0 can be handled correctly. This is especially handy, if functions with a variable amount of outputs are wrapped by `delayed`. E.g. a trivial example: `delayed(lambda *args: args, nout=len(vals))(*vals)`

#### Core

- Refactor core byte ingest (GH#1768), (GH#1774)
- Improve import time (GH#1833)

## 4.23.14 0.12.0 / 2016-11-03

### DataFrame

- Return a series when functions given to `dataframe.map_partitions` return scalars (GH#1515)
- Fix type size inference for series (GH#1513)
- `dataframe.DataFrame.categorize` no longer includes missing values in the categories. This is for compatibility with a pandas change (GH#1565)
- Fix head parser error in `dataframe.read_csv` when some lines have quotes (GH#1495)
- Add `dataframe.reduction` and `series.reduction` methods to apply generic row-wise reduction to dataframes and series (GH#1483)
- Add `dataframe.select_dtypes`, which mirrors the pandas method (GH#1556)
- `dataframe.read_hdf` now supports reading Series (GH#1564)
- Support Pandas 0.19.0 (GH#1540)
- Implement `select_dtypes` (GH#1556)
- String accessor works with indexes (GH#1561)
- Add pipe method to `dask.dataframe` (GH#1567)
- Add `indicator` keyword to `merge` (GH#1575)
- Support Series in `read_hdf` (GH#1575)
- Support Categories with missing values (GH#1578)
- Support inplace operators like `df.x += 1` (GH#1585)
- Str accessor passes through args and kwargs (GH#1621)
- Improved groupby support for single-machine multiprocessing scheduler (GH#1625)
- Tree reductions (GH#1663)
- Pivot tables (GH#1665)
- Add `clip` (GH#1667), `align` (GH#1668), `combine_first` (GH#1725), and `any/all` (GH#1724)
- Improved handling of divisions on dask-pandas merges (GH#1666)
- Add `groupby.aggregate` method (GH#1678)
- Add `dd.read_table` function (GH#1682)
- Improve support for multi-level columns (GH#1697) (GH#1712)
- Support 2d indexing in `loc` (GH#1726)
- Extend `resample` to include DataFrames (GH#1741)
- Support `dask.array` ufuncs on `dask.dataframe` objects (GH#1669)

### Array

- Add information about how `dask.array` chunks argument work (GH#1504)
- Fix field access with non-scalar fields in `dask.array` (GH#1484)
- Add `concatenate=` keyword to `atop` to concatenate chunks of contracted dimensions

- Optimized slicing performance (GH#1539) (GH#1731)
- Extend `atop` with a `concatenate=` (GH#1609) `new_axes=` (GH#1612) and `adjust_chunks=` (GH#1716) keywords
- Add `clip` (GH#1610) `swapaxes` (GH#1611) `round` (GH#1708) `repeat`
- Automatically align chunks in `atop`-backed operations (GH#1644)
- Cull `dask.arrays` on slicing (GH#1709)

## Bag

- Fix issue with callables in `bag.from_sequence` being interpreted as tasks (GH#1491)
- Avoid non-lazy memory use in reductions (GH#1747)

## Administration

- Added changelog (GH#1526)
- Create new threadpool when operating from thread (GH#1487)
- Unify example documentation pages into one (GH#1520)
- Add versioneer for git-commit based versions (GH#1569)
- Pass through `node_attr` and `edge_attr` keywords in dot visualization (GH#1614)
- Add continuous testing for Windows with Appveyor (GH#1648)
- Remove use of `multiprocessing.Manager` (GH#1653)
- Add global optimizations keyword to `compute` (GH#1675)
- Micro-optimize `get_dependencies` (GH#1722)

## 4.23.15 0.11.0 / 2016-08-24

### Major Points

DataFrames now enforce knowing full metadata (columns, dtypes) everywhere. Previously we would operate in an ambiguous state when functions lost dtype information (such as `apply`). Now all dataframes always know their dtypes and raise errors asking for information if they are unable to infer (which they usually can). Some internal attributes like `_pd` and `_pd_nonempty` have been moved.

The internals of the distributed scheduler have been refactored to transition tasks between explicit states. This improves resilience, reasoning about scheduling, plugin operation, and logging. It also makes the scheduler code easier to understand for newcomers.

### Breaking Changes

- The `distributed.s3` and `distributed.hdfs` namespaces are gone. Use protocols in normal methods like `read_text('s3://...')` instead.
- `Dask.array.reshape` now errs in some cases where previously it would have create a very large number of tasks

### 4.23.16 0.10.2 / 2016-07-27

- More Dataframe shuffles now work in distributed settings, ranging from setting-index to hash joins, to sorted joins and groupbys.
- Dask passes the full test suite when run when under in Python's optimized-OO mode.
- On-disk shuffles were found to produce wrong results in some highly-concurrent situations, especially on Windows. This has been resolved by a fix to the partd library.
- Fixed a growth of open file descriptors that occurred under large data communications
- Support ports in the `--bokeh-whitelist` option of `dask-scheduler` to better routing of web interface messages behind non-trivial network settings
- Some improvements to resilience to worker failure (though other known failures persist)
- You can now start an IPython kernel on any worker for improved debugging and analysis
- Improvements to `dask.dataframe.read_hdf`, especially when reading from multiple files and docs

### 4.23.17 0.10.0 / 2016-06-13

#### Major Changes

- This version drops support for Python 2.6
- Conda packages are built and served from conda-forge
- The `dask.distributed` executables have been renamed from `dfoo` to `dask-foo`. For example `dscheduler` is renamed to `dask-scheduler`
- Both `Bag` and `DataFrame` include a preliminary distributed shuffle.

#### Bag

- Add task-based shuffle for distributed groupbys
- Add `accumulate` for cumulative reductions

#### DataFrame

- Add a task-based shuffle suitable for distributed joins, `groupby-applys`, and `set_index` operations. The single-machine shuffle remains untouched (and much more efficient.)
- Add support for new Pandas rolling API with improved communication performance on distributed systems.
- Add `groupby.std/var`
- Pass through S3/HDFS storage options in `read_csv`
- Improve categorical partitioning
- Add `eval`, `info`, `isnull`, `notnull` for dataframes

## Distributed

- Rename executables like dscheduler to dask-scheduler
- Improve scheduler performance in the many-fast-tasks case (important for shuffling)
- Improve work stealing to be aware of expected function run-times and data sizes. This drastically increases the breadth of algorithms that can be efficiently run on the distributed scheduler without significant user expertise.
- Support maximum buffer sizes in streaming queues
- Improve Windows support when using the Bokeh diagnostic web interface
- Support compression of very-large-bytestrings in protocol
- Support clean cancellation of submitted futures in Joblib interface

## Other

- All dask-related projects (dask, distributed, s3fs, hdfs, partd) are now building conda packages on conda-forge.
- Change credential handling in s3fs to only pass around delegated credentials if explicitly given secret/key. The default now is to rely on managed environments. This can be changed back by explicitly providing a keyword argument. Anonymous mode must be explicitly declared if desired.

## 4.23.18 0.9.0 / 2016-05-11

### API Changes

- `dask.do` and `dask.value` have been renamed to `dask.delayed`
- `dask.bag.from_filenames` has been renamed to `dask.bag.read_text`
- All S3/HDFS data ingest functions like `db.from_s3` or `distributed.s3.read_csv` have been moved into the plain `read_text`, `read_csv` functions, which now support protocols, like `dd.read_csv('s3://bucket/keys*.csv')`

### Array

- Add support for `scipy.LinearOperator`
- Improve optional locking to on-disk data structures
- Change `rechunk` to expose the intermediate chunks

### Bag

- Rename `from_filename`'s to `read_text`
- Remove `from_s3` in favor of `read_text('s3://...')`

## DataFrame

- Fixed numerical stability issue for correlation and covariance
- Allow no-hash `from_pandas` for speedy round-trips to and from-pandas objects
- Generally reengineered `read_csv` to be more in line with Pandas behavior
- Support fast `set_index` operations for sorted columns

## Delayed

- Rename `do/value` to `delayed`
- Rename `to/from_imperative` to `to/from_delayed`

## Distributed

- Move `s3` and `hdfs` functionality into the dask repository
- Adaptively oversubscribe workers for very fast tasks
- Improve PyPy support
- Improve work stealing for unbalanced workers
- Scatter data efficiently with `tree-scatters`

## Other

- Add `lzma/xz` compression support
- Raise a warning when trying to split unsplitable compression types, like `gzip` or `bz2`
- Improve hashing for single-machine shuffle operations
- Add new `callback` method for start state
- General performance tuning

## 4.23.19 0.8.1 / 2016-03-11

### Array

- Bugfix for range slicing that could periodically lead to incorrect results.
- Improved support and resiliency of `arg` reductions (`argmin`, `argmax`, etc.)

### Bag

- Add `zip` function

## DataFrame

- Add `corr` and `cov` functions
- Add `melt` function
- Bugfixes for `io` to `bcolz` and `hdf5`

### 4.23.20 0.8.0 / 2016-02-20

## Array

- Changed default array reduction split from 32 to 4
- Linear algebra, `tril`, `triu`, `LU`, `inv`, `cholesky`, `solve`, `solve_triangular`, `eye`, `lstsq`, `diag`, `corrcoef`.

## Bag

- Add tree reductions
- Add range function
- `drop_from_hdfs` function (better functionality now exists in `hdfs3` and distributed projects)

## DataFrame

- Refactor `dask.dataframe` to include a full empty pandas dataframe as metadata. Drop the `.columns` attribute on `Series`
- Add `Series` categorical accessor, `series.nunique`, drop the `.columns` attribute for series.
- `read_csv` fixes (multi-column `parse_dates`, integer column names, etc. )
- Internal changes to improve graph serialization

## Other

- Documentation updates
- Add `from_imperative` and `to_imperative` functions for all collections
- Aesthetic changes to profiler plots
- Moved the dask project to a new dask organization

### 4.23.21 0.7.6 / 2016-01-05

## Array

- Improve thread safety
- Tree reductions
- Add `view`, `compress`, `hstack`, `dstack`, `vstack` methods
- `map_blocks` can now remove and add dimensions

## DataFrame

- Improve thread safety
- Extend sampling to include replacement options

## Imperative

- Removed optimization passes that fused results.

## Core

- Removed `dask.distributed`
- Improved performance of blocked file reading
- Serialization improvements
- Test Python 3.5

### 4.23.22 0.7.4 / 2015-10-23

This was mostly a bugfix release. Some notable changes:

- Fix minor bugs associated with the release of numpy 1.10 and pandas 0.17
- Fixed a bug with random number generation that would cause repeated blocks due to the birthday paradox
- Use locks in `dask.dataframe.read_hdf` by default to avoid concurrency issues
- Change `dask.get` to point to `dask.async.get_sync` by default
- Allow visualization functions to accept general graphviz graph options like `rankdir='LR'`
- Add `reshape` and `ravel` to `dask.array`
- Support the creation of `dask.arrays` from `dask.imperative` objects

## Deprecation

This release also includes a deprecation warning for `dask.distributed`, which will be removed in the next version.

Future development in distributed computing for dask is happening here: <https://distributed.readthedocs.io> . General feedback on that project is most welcome from this community.

### 4.23.23 0.7.3 / 2015-09-25

## Diagnostics

- A utility for profiling memory and cpu usage has been added to the `dask.diagnostics` module.

## DataFrame

This release improves coverage of the pandas API. Among other things it includes `nunique`, `nlargest`, `quantile`. Fixes encoding issues with reading non-ascii csv files. Performance improvements and bug fixes with `resample`. More flexible `read_hdf` with globbing. And many more. Various bug fixes in `dask.imperative` and `dask.bag`.

### 4.23.24 0.7.0 / 2015-08-15

## DataFrame

This release includes significant bugfixes and alignment with the Pandas API. This has resulted both from use and from recent involvement by Pandas core developers.

- New operations: query, rolling operations, drop
- Improved operations: quantiles, arithmetic on full dataframes, dropna, constructor logic, merge/join, elemwise operations, groupby aggregations

## Bag

- Fixed a bug in fold where with a null default argument

## Array

- New operations: `da.fft` module, `da.image.imread`

## Infrastructure

- The array and dataframe collections create graphs with deterministic keys. These tend to be longer (hash strings) but should be consistent between computations. This will be useful for caching in the future.
- All collections (Array, Bag, DataFrame) inherit from common subclass

### 4.23.25 0.6.1 / 2015-07-23

## Distributed

- Improved (though not yet sufficient) resiliency for `dask.distributed` when workers die

## DataFrame

- Improved writing to various formats, including `to_hdf`, `to_castra`, and `to_csv`
- Improved creation of dask DataFrames from dask Arrays and Bags
- Improved support for categoricals and various other methods

## Array

- Various bug fixes
- Histogram function

## Scheduling

- Added tie-breaking ordering of tasks within parallel workloads to better handle and clear intermediate results

## Other

- Added the `dask.do` function for explicit construction of graphs with normal python code
- Traded `pydot` for `graphviz` library for graph printing to support Python3
- There is also a gitter chat room and a stackoverflow tag

## 4.24 Dask Cheat Sheet

The 300KB pdf `dask cheat sheet` is a single page summary of all the most important information about using dask.

## 4.25 Presentations On Dask

- PLOTCON 2016, December 2016
  - Visualizing Distributed Computations with Dask and Bokeh
- PyData DC, October 2016
  - Using Dask for Parallel Computing in Python
- SciPy 2016, July 2016
  - Dask Parallel and Distributed Computing
- PyData NYC, December 2015
  - Dask Parallelizing NumPy and Pandas through Task Scheduling
- PyData Seattle, August 2015
  - Dask: out of core arrays with task scheduling
- SciPy 2015, July 2015
  - Dask Out of core NumPy:Pandas through Task Scheduling

## 4.26 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.

### 4.26.1 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.

### 4.26.2 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>

### 4.26.3 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.

### 4.26.4 Development Environment

#### Download code

Clone the main dask git repository (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
```

## 4.26.5 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 Parameters, 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
```

Docstring testing requires `graphviz` to be installed. This can be done via:

```
conda install -y graphviz
```

## Style

Dask verifies style uniformity with the `flake8` tool.:

```
pip install flake8
flake8 dask
```

## Changelog

Every significant code contribution should be listed in the *Changelog* under the corresponding version. When submitting a Pull Request in Github please add to that file explaining what was added/modified.

### 4.26.6 Contributing to Documentation

Dask uses [Sphinx](http://readthedocs.org) 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.

## 4.27 Frequently Asked Questions

We maintain most Q&A on [Stack Overflow](https://stackoverflow.com) under the `#Dask` tag. You may find the questions there useful to you.

### 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` by providing a `get=` keyword to the `compute` method:

```
my_array.compute(get=dask.get)
```

### 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
```

## 4.28 Comparison to Spark

[Apache Spark](#) is a popular distributed computing tool for tabular datasets that is growing to become a dominant name in Big Data analysis today. 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.

### 4.28.1 Summary

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. It couples with other libraries like Pandas or Scikit-Learn to achieve high-level functionality.

- **Language**
  - Spark is written in Scala with some support for Python and R. It interoperates well with other JVM code.
  - Dask is written in Python and only really supports Python. It interoperates well with C/C++/Fortran/LLVM or other natively compiled code linked through Python.
- **Ecosystem**
  - Spark is an all-in-one project that has inspired its own ecosystem. It integrates well with many other Apache projects.
  - Dask is a component of the larger Python ecosystem. It couples with and enhances other libraries like NumPy, Pandas, and Scikit-Learn.
- **Age and Trust**
  - Spark is older (since 2010) and has become a dominant and well-trusted tool in the Big Data enterprise world.
  - Dask is younger (since 2014) and is an extension of the well trusted NumPy/Pandas/Scikit-learn/Jupyter stack.
- **Scope**

- Spark is more focused on traditional business intelligence operations like SQL and lightweight machine learning.
- Dask is applied more generally both to business intelligence applications, as well as a number of scientific and custom business situations

- **Internal Design**

- Spark’s internal model is higher level, providing good high level optimizations on uniformly applied computations, but lacking flexibility for more complex algorithms or ad-hoc systems. It is fundamentally an extension of the Map-Shuffle-Reduce paradigm.
- Dask’s internal model is lower level, and so lacks high level optimizations, but is able to implement more sophisticated algorithms and build more complex bespoke systems. It is fundamentally based on generic task scheduling.

- **Scale**

- Spark scales from a single node to thousand-node clusters
- Dask scales from a single node to thousand-node clusters

- **APIs**

- **Dataframes**

- \* Spark dataframe has its own API and memory model. It also implements a large subset of the SQL language. Spark includes a high-level query optimizer for complex queries.
- \* Dask.dataframe reuses the Pandas API and memory model. It implements neither SQL nor a query optimizer. It is able to do random access, efficient time series operations, and other Pandas-style indexed operations.

- **Machine Learning**

- \* Spark MLLib is a cohesive project with support for common operations that are easy to implement with Spark’s Map-Shuffle-Reduce style system. People considering MLLib might also want to consider *other* JVM-based machine learning libraries like H2O, which may have better performance.
- \* Dask relies on and interoperates with existing libraries like Scikit-Learn and XGBoost. These can be more familiar or higher performance, but generally results in a less-cohesive whole. See the [dask-ml](#) project for integrations.

- **Arrays**

- \* Spark does not include support for multi-dimensional arrays natively (this would be challenging given their computation model) although some support for two-dimensional matrices may be found in MLLib. People may also want to look at the [Thunder](#) project.
- \* Dask fully supports the NumPy model for *scalable multi-dimensional arrays*.

- **Streaming**

- \* Spark’s support for streaming data is first-class and integrates well into their other APIs. It follows a mini-batch approach. This provides decent performance on large uniform streaming operations
- \* Dask provides a real-time futures interface that is lower-level than Spark streaming. This enables more creative and complex use-cases, but requires more work than Spark streaming.

- **Graphs / complex networks**

- \* Spark provides GraphX, a library for graph processing

- \* Dask provides no such library

- **Custom parallelism**

- \* Spark generally expects users to compose computations out of their high-level primitives (map, reduce, groupby, join, ...). It is also possible to extend Spark through subclassing RDDs, although this is rarely done.
- \* Dask allows you to specify arbitrary task graphs for more complex and custom systems that are not part of the standard set of collections.

## 4.28.2 Reasons you might choose Spark

- You prefer Scala or the SQL language
- You have mostly JVM infrastructure and legacy systems
- You want an established and trusted solution for business
- You are mostly doing business analytics with some lightweight machine learning
- You want an all-in-one solution

## 4.28.3 Reasons you might choose Dask

- You prefer Python or native code, or have large legacy code bases that you do not want to entirely rewrite
- Your use case is complex or does not cleanly fit the Spark computing model
- You want a lighter-weight transition from single-machine computing to cluster computing
- You want to interoperate with other technologies and don't mind installing multiple packages

## 4.28.4 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.

### 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”.

## 4.28.5 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](#).

## 4.29 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.

### 4.29.1 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.

## 4.29.2 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.

## 4.29.3 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
```

## 4.29.4 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.

### 4.29.5 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.

## 4.30 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.*

<code>read_bytes(urlpath[, delimiter, not_zero, ...])</code>	Convert path to a list of delayed values
<code>open_files(urlpath[, compression, mode, ...])</code>	Given path return <code>dask.delayed</code> file-like objects

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`). 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.

### 4.30.1 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.

### 4.30.2 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
```

The various back-ends accept optional extra keywords, detailing authentication and other parameters, see [remote data services](#)

### 4.30.3 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.

### 4.30.4 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'\n'` on which to split blocks of bytes.

**not\_zero: bool**

Force seek of start-of-file delimiter, discarding header.

**blocksize: int (=128MB)**

Chunk size in bytes

**compression: string or None**

String like `'gzip'` or `'xz'`. Must support efficient random access.

**sample: bool or int**

Whether or not to return a header sample. If an integer is given it is used as sample size, otherwise the default sample size is 10kB.

**\*\*kwargs: dict**

Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

**Returns** A 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, mode='rb', encoding='utf8', errors=None, name_function=None, num=1, **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.

#### `mode`: 'rb', 'wt', etc.

#### `encoding`: str

For text mode only

#### `errors`: None or str

Passed to `TextIOWrapper` in text mode

#### `name_function`: function or None

if opening a set of files for writing, those files do not yet exist, so we need to generate their names by formatting the `urlpath` for each sequence number

#### `num`: int [1]

if writing mode, number of files we expect to create (passed to `name+function`)

#### `**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')
```

## 4.31 Remote Data Services

As described in Section *Internal Data Ingestion*, various user-facing functions (such as `dataframe.read_csv`, `dataframe.read_parquet`, `bag.read_text`) and lower level byte-manipulating functions may point to data that lives not on the local storage of the workers, but on a remote system such as Amazon S3.

In this section we describe how to use the various known back-end storage systems. Below we give some details for interested developers on how further storage back-ends may be provided for dask's use.

### 4.31.1 Known Storage Implementations

When specifying a storage location, a URL should be provided using the general form `protocol://path/to/data`. If no protocol is provided, the local file-system is assumed (same as `file://`). Two methods exist for passing parameters to the backend file-system driver: extending the URL to include username, password, server, port, etc.; and providing `storage_options`, a dictionary of parameters to pass on. Examples:

```
df = dd.read_csv('hdfs://user@server:port/path/*.csv')

df = dd.read_parquet('s3://bucket/path',
                    storage_options={'anon': True, 'use_ssl': False})
```

Further details on how to provide configuration for each backend is listed next.

Each back-end has additional installation requirements and may not be available at runtime. The dictionary `dask.bytes.core._filesystems` contains the currently available file-systems. Some require appropriate imports before use.

The following list gives the protocol shorthands and the back-ends they refer to:

- `file:` - the local file system, default in the absence of any protocol
- `hdfs:` - Hadoop Distributed File System, a for resilient, replicated files within a cluster, using the library `hdfs3`
- `s3:` - Amazon S3 remote binary store, often used with Amazon EC2, using the library `s3fs`
- `gcs:` or `gs:` - Google Cloud Storage, typically used with Google Compute resource, using `gcsfs` (in development)

### Local File System

Local files are always accessible, and all parameters passed as part of the URL (beyond the path itself) or with the `storage_options` dictionary will be ignored.

This is the default back-end, and the one used if no protocol is passed at all.

We assume here that each worker has access to the same file-system - either the workers are co-located on the same machine, or a network file system is mounted and referenced at the same path location for every worker node.

Locations specified relative to the current working directory will, in general, be respected (as they would be with the built-in python `open`), but this may fail in the case that the client and worker processes do not necessarily have the same working directory.

### HDFS

The Hadoop File System (HDFS) is a widely deployed, distributed, data-local file system written in Java. This file system backs many clusters running Hadoop and Spark.

Within dask, HDFS is only available when the module `distributed.hdfs` is explicitly imported, since the usage of HDFS usually only makes sense in a cluster setting. The distributed scheduler will prefer to allocate tasks which read from HDFS to machines which have local copies of the blocks required for their work, where possible.

By default, `hdfs3` attempts to read the default server and port from local Hadoop configuration files on each node, so it may be that no configuration is required. However, the server, port and user can be passed as part of the url: `hdfs://user:pass@server:port/path/to/data`.

The following parameters may be passed to `hdfs3` using `storage_options`:

- `host, port, user`: basic authentication
- `ticket_cache, token`: kerberos authentication
- `pars`: dictionary of further parameters (e.g., for [high availability](#))

Important environment variables:

- `HADOOP_CONF_DIR` or `HADOOP_INSTALL`: directory containing `core-site.xml` and/or `hdfs-site.xml`, containing configuration information
- `LIBHDFS3_CONF`: location of a specific xml file with configuration for the `libhdfs3` client; this may be the same as one of the files above. *Short circuit* reads should be defined in this file ([see here](#))

## S3

Amazon S3 (Simple Storage Service) is a web service offered by Amazon Web Services.

The S3 back-end will be available to dask is `s3fs` is importable when dask is imported.

Authentication for S3 is provided by the underlying library `boto3`. As described in the [auth docs](#) this could be achieved by placing credentials files in one of several locations on each node: `~/.aws/credentials`, `~/.aws/config`, `/etc/boto.cfg` and `~/.boto`. Alternatively, for nodes located within Amazon EC2, IAM roles can be set up for each node, and then no further configuration is required. The final authentication option, is for user credentials can be passed directly in the URL (`s3://keyID:keySecret/bucket/key/name`) or using `storage_options`. In this case, however, the key/secret will be passed to all workers in-the-clear, so this method is only recommended on well-secured networks.

The following parameters may be passed to `s3fs` using `storage_options`:

- `anon`: whether access should be anonymous (default False)
- `key, secret`: for user authentication
- `token`: if authentication has been done with some other S3 client
- `use_ssl`: whether connections are encrypted and secure (default True)
- `client_kwargs`: dict passed to the `boto3` client, with keys such as `region_name`, `endpoint_url`
- `requester_pays`: set True if the authenticated user will assume transfer costs, which is required by some providers of bulk data
- `default_block_size`, `default_fill_cache`: these are not of particular interest to dask users, as they concern the behaviour of the buffer between successive reads
- `kwargs`: other parameters are passed to the `boto3 Session` object, such as `profile_name`, to pick one of the authentication sections from the configuration files referred to above ([see here](#))

## Google Cloud Storage

(gcsfs is in early development, expect the details here to change)

Google Cloud Storage is a RESTful online file storage web service for storing and accessing data on Google's infrastructure.

The GCS backend will be available only after importing `gcsfs`. The protocol identifiers `gcs` and `gs` are identical in their effect.

Authentication for GCS is based on OAuth2, and designed for user verification. Interactive authentication is available when `token==None` using the local browser, or by using `gcloud` to produce a JSON token file and passing that. In either case, `gcsfs` stores a cache of tokens in a local file, so subsequent authentication will not be necessary.

At the time of writing, `gcsfs.GCSFileSystem` instances pickle including the auth token, so sensitive information is passed between nodes of a dask distributed cluster. This will be changed to allow the use of either local JSON or pickle files for storing tokens and authenticating on each node automatically, instead of passing around an authentication token, similar to S3, above.

Every use of GCS requires the specification of a project to run within - if the project is left empty, the user will not be able to perform any bucket-level operations. The project can be defined using the variable `GCSFS_DEFAULT_PROJECT` in the environment of every worker, or by passing something like the following

```
dd.read_parquet('gs://bucket/path', storage_options={'project': 'myproject'})
```

Possible additional storage options:

- `access`: 'read\_only', 'read\_write', 'full\_control', access privilege level (note that the token cache uses a separate token for each level)
- `token`: either an actual dictionary of a google token, or location of a JSON file created by `gcloud`.

### 4.31.2 Developer API

The prototype for any file-system back-end can be found in `bytes.local.LocalFileSystem`. Any new implementation should provide the same API, and make itself available as a protocol to dask. For example, the following would register the protocol "myproto", described by the implementation class `MyProtoFileSystem`. URLs of the form `myproto://` would thereafter be dispatched to the methods of this class.

```
dask.bytes.core._filesystems['myproto'] = MyProtoFileSystem
```

For a more complicated example, users may wish to also see `dask.bytes.s3.DaskS3FileSystem`.

```
class dask.bytes.local.LocalFileSystem(**storage_options)  
    API spec for the methods a filesystem
```

A filesystem must provide these methods, if it is to be registered as a backend for dask.

Implementation for local disc

```
glob (path)
```

For a template path, return matching files

```
makedirs (path)
```

Make any intermediate directories to make path writable

```
open (path, mode='rb', **kwargs)
```

Make a file-like object

**Parameters** mode: string

normally “rb”, “wb” or “ab” or other.

**kwargs: key-value**

Any other parameters, such as buffer size. May be better to set these on the filesystem instance, to apply to all files created by it. Not used for local.

**size** (*path*)

Size in bytes of the file at path

**ukey** (*path*)

Unique identifier, so we can tell if a file changed

## 4.32 Custom Collections

For many problems the built-in dask collections (`dask.array`, `dask.dataframe`, `dask.bag`, and `dask.delayed`) are sufficient. For cases where they aren’t it’s possible to create your own dask collections. Here we describe the required methods to fulfill the dask collection interface.

**Warning:** The custom collection API is experimental and subject to change without going through a deprecation cycle.

---

**Note:** This is considered an advanced feature. For most cases the built-in collections are probably sufficient.

---

Before reading this you should read and understand:

- *overview*
- *graph specification*
- *custom graphs*

### Contents

- *Description of the dask collection interface*
- *How this interface is used to implement the core dask methods*
- *How to add the core methods to your class*
- *Example Dask Collection*
- *How to check if something is a dask collection*
- *How to make tokenize work with your collection*

### 4.32.1 The Dask Collection Interface

To create your own dask collection, you need to fulfill the following interface. Note that there is no required base class.

It’s recommended to also read *Internals of the Core Dask Methods* to see how this interface is used inside dask.

`__dask_graph__` (*self*)

The dask graph.

**Returns** `dsk` : MutableMapping, None

The dask graph. If `None`, this instance will not be interpreted as a dask collection, and none of the remaining interface methods will be called.

`__dask_keys__` (*self*)

The output keys for the dask graph.

**Returns** `keys` : list

A possibly nested list of keys that represent the outputs of the graph. After computation, the results will be returned in the same layout, with the keys replaced with their corresponding outputs.

**static** `__dask_optimize__` (*dsk*, *keys*, *\*\*kwargs*)

Given a graph and keys, return a new optimized graph.

This method can be either a `staticmethod` or a `classmethod`, but not an `instancemethod`.

Note that graphs and keys are merged before calling `__dask_optimize__`; as such the graph and keys passed to this method may represent more than one collection sharing the same optimize method.

If not implemented, defaults to returning the graph unchanged.

**Parameters** `dsk` : MutableMapping

The merged graphs from all collections sharing the same `__dask_optimize__` method.

**keys** : list

A list of the outputs from `__dask_keys__` from all collections sharing the same `__dask_optimize__` method.

**\*\*kwargs**

Extra keyword arguments forwarded from the call to `compute` or `persist`. Can be used or ignored as needed.

**Returns** `optimized_dsk` : MutableMapping

The optimized dask graph.

**static** `__dask_scheduler__` (*dsk*, *keys*, *\*\*kwargs*)

The default scheduler `get` to use for this object.

Usually attached to the class as a `staticmethod`, e.g.

```
>>> import dask.threaded
>>> class MyCollection(object):
...     # Use the threaded scheduler by default
...     __dask_scheduler__ = staticmethod(dask.threaded.get)
```

`__dask_postcompute__` (*self*)

Return the finalizer and (optional) extra arguments to convert the computed results into their in-memory representation.

Used to implement `dask.compute`.

**Returns** `finalize` : callable

A function with the signature `finalize(results, *extra_args)`. Called with the computed results in the same structure as the corresponding keys from `__dask_keys__`, as well as any extra arguments as specified in `extra_args`. Should perform any necessary finalization before returning the corresponding in-memory collection from `compute`. For example, the `finalize` function for

`dask.array.Array` concatenates all the individual array chunks into one large numpy array, which is then the result of `compute`.

**extra\_args** : tuple

Any extra arguments to pass to `finalize` after `results`. If no extra arguments should be an empty tuple.

**\_\_dask\_postpersist\_\_** (*self*)

Return the `rebuilder` and (optional) extra arguments to rebuild an equivalent dask collection from a persisted graph.

Used to implement `dask.persist`.

**Returns** `rebuild` : callable

A function with the signature `rebuild(dsk, *extra_args)`. Called with a persisted graph containing only the keys and results from `__dask_keys__`, as well as any extra arguments as specified in `extra_args`. Should return an equivalent dask collection with the same keys as `self`, but with the results already computed. For example, the `rebuild` function for `dask.array.Array` is just the `__init__` method called with the new graph but the same metadata.

**extra\_args** : tuple

Any extra arguments to pass to `rebuild` after `dsk`. If no extra arguments should be an empty tuple.

---

**Note:** It's also recommended to define `__dask_tokenize__`, see *Implementing Deterministic Hashing*.

---

## 4.32.2 Internals of the Core Dask Methods

Dask has a few *core* functions (and corresponding methods) that implement common operations:

- `compute`: convert one or more dask collections into their in-memory counterparts
- `persist`: convert one or more dask collections into equivalent dask collections with their results already computed and cached in memory.
- `optimize`: convert one or more dask collections into equivalent dask collections sharing one large optimized graph.
- `visualize`: given one or more dask collections, draw out the graph that would be passed to the scheduler during a call to `compute` or `persist`

Here we briefly describe the internals of these functions to illustrate how they relate to the above interface.

### Compute

The operation of `compute` can be broken into three stages:

#### 1. Graph Merging & Optimization

First the individual collections are converted to a single large graph and nested list of keys. How this happens depends on the value of the `optimize_graph` keyword, which each function takes:

- If `optimize_graph` is `True` (default) then the collections are first grouped by their `__dask_optimize__` methods. All collections with the same `__dask_optimize__` method

have their graphs merged and keys concatenated, and then a single call to each respective `__dask_optimize__` is made with the merged graphs and keys. The resulting graphs are then merged.

- If `optimize_graph` is `False` then all the graphs are merged and all the keys concatenated.

After this stage there is a single large graph and nested list of keys which represents all the collections.

## 2. Computation

After the graphs are merged and any optimizations performed, the resulting large graph and nested list of keys are passed on to the scheduler. The scheduler to use is chosen as follows:

- If a `get` function is specified directly as a keyword, use that.
- Otherwise, if a global scheduler is set, use that.
- Otherwise fall back to the default scheduler for the given collections. Note that if all collections don't share the same `__dask_scheduler__` then an error will be raised.

Once the appropriate scheduler `get` function is determined, it's called with the merged graph, keys, and extra keyword arguments. After this stage `results` is a nested list of values. The structure of this list mirrors that of `keys`, with each key substituted with its corresponding result.

## 3. Postcompute

After the results are generated the output values of `compute` need to be built. This is what the `__dask_postcompute__` method is for. `__dask_postcompute__` returns two things:

- A `finalize` function, which takes in the results for the corresponding keys
- A tuple of extra arguments to pass to `finalize` after the results

To build the outputs, the list of collections and results is iterated over, and the finalizer for each collection is called on its respective results.

In pseudocode this process looks like:

```
def compute(*collections, **kwargs):
    # 1. Graph Merging & Optimization
    # -----
    if kwargs.pop('optimize_graph', True):
        # If optimization is turned on, group the collections by
        # optimization method, and apply each method only once to the merged
        # sub-graphs.
        optimization_groups = groupby_optimization_methods(collections)
        graphs = []
        for optimize_method, cols in optimization_groups:
            # Merge the graphs and keys for the subset of collections that
            # share this optimization method
            sub_graph = merge_graphs([x.__dask_graph__() for x in cols])
            sub_keys = [x.__dask_keys__() for x in cols]
            # kwargs are forwarded to ``__dask_optimize__`` from compute
            optimized_graph = optimize_method(sub_graph, sub_keys, **kwargs)
            graphs.append(optimized_graph)
        graph = merge_graphs(graphs)
    else:
        graph = merge_graphs([x.__dask_graph__() for x in collections])
        # Keys are always the same
        keys = [x.__dask_keys__() for x in collections]

    # 2. Computation
    # -----
    # Determine appropriate get function based on collections, global
```

```

# settings, and keyword arguments
get = determine_get_function(collections, **kwargs)
# Pass the merged graph, keys, and kwargs to `get`
results = get(graph, keys, **kwargs)

# 3. Postcompute
# -----
output = []
# Iterate over the results and collections
for res, collection in zip(results, collections):
    finalize, extra_args = collection.__dask_postcompute__()
    out = finalize(res, **extra_args)
    output.append(out)

# `dask.compute` always returns tuples
return tuple(output)

```

## Persist

Persist is very similar to `compute`, except for how the return values are created. It too has three stages:

### 1. Graph Merging & Optimization

Same as in `compute`.

### 2. Computation

Same as in `compute`, except in the case of the distributed scheduler, where the values in `results` are futures instead of values.

### 3. Postpersist

Similar to `__dask_postcompute__`, `__dask_postpersist__` is used to rebuild values in a call to `persist`. `__dask_postpersist__` returns two things:

- A rebuild function, which takes in a persisted graph. The keys of this graph are the same as `__dask_keys__` for the corresponding collection, and the values are computed results (for the single machine scheduler) or futures (for the distributed scheduler).
- A tuple of extra arguments to pass to rebuild after the graph

To build the outputs of `persist`, the list of collections and results is iterated over, and the rebuild function for each collection is called on the graph for its respective results.

In pseudocode this looks like:

```

def persist(*collections, **kwargs):
    # 1. Graph Merging & Optimization
    # -----
    # **Same as in compute**
    graph = ...
    keys = ...

    # 2. Computation
    # -----
    # **Same as in compute**
    results = ...

    # 3. Postpersist
    # -----

```

```
output = []
# Iterate over the results and collections
for res, collection in zip(results, collections):
    # res has the same structure as keys
    keys = collection.__dask_keys__()
    # Get the computed graph for this collection.
    # Here flatten converts a nested list into a single list
    subgraph = {k: r for (k, r) in zip(flatten(keys), flatten(res))}

    # Rebuild the output dask collection with the computed graph
    rebuild, extra_args = collection.__dask_postpersist__()
    out = rebuild(subgraph, *extra_args)

    output.append(out)

# dask.persist always returns tuples
return tuple(output)
```

## Optimize

The operation of optimize can be broken into two stages:

### 1. Graph Merging & Optimization

Same as in compute.

### 2. Rebuilding

Similar to persist, the rebuild function and arguments from `__dask_postpersist__` are used to reconstruct equivalent collections from the optimized graph.

In pseudocode this looks like:

```
def optimize(*collections, **kwargs):
    # 1. Graph Merging & Optimization
    # -----
    # **Same as in compute**
    graph = ...

    # 2. Rebuilding
    # -----
    # Rebuild each dask collection using the same large optimized graph
    output = []
    for collection in collections:
        rebuild, extra_args = collection.__dask_postpersist__()
        out = rebuild(graph, *extra_args)
        output.append(out)

    # dask.optimize always returns tuples
    return tuple(output)
```

## Visualize

Visualize is the simplest of the 4 core functions. It only has two stages:

### 1. Graph Merging & Optimization

Same as in compute

## 2. Graph Drawing

The resulting merged graph is drawn using graphviz and output to the specified file.

In pseudocode this looks like:

```
def visualize(*collections, **kwargs):
    # 1. Graph Merging & Optimization
    # -----
    # **Same as in compute**
    graph = ...

    # 2. Graph Drawing
    # -----
    # Draw the graph with graphviz's `dot` tool and return the result.
    return dot_graph(graph, **kwargs)
```

### 4.32.3 Adding the Core Dask Methods to Your Class

Defining the above interface will allow your object to be used by the core dask functions (`dask.compute`, `dask.persist`, `dask.visualize`, etc...). To add corresponding method versions of these subclass from `dask.base.DaskMethodsMixin`, which adds implementations of `compute`, `persist`, and `visualize` based on the interface above.

### 4.32.4 Example Dask Collection

Here we create a dask collection representing a tuple. Every element in the tuple is represented as a task in the graph. Note that this is for illustration purposes only - the same user experience could be done using normal tuples with elements of `dask.delayed`.

```
# Saved as dask_tuple.py
from dask.base import DaskMethodsMixin
from dask.optimization import cull

# We subclass from DaskMethodsMixin to add common dask methods to our
# class. This is nice but not necessary for creating a dask collection.
class Tuple(DaskMethodsMixin):
    def __init__(self, dsk, keys):
        # The init method takes in a dask graph and a set of keys to use
        # as outputs.
        self._dsk = dsk
        self._keys = keys

    def __dask_graph__(self):
        return self._dsk

    def __dask_keys__(self):
        return self._keys

    @staticmethod
    def __dask_optimize__(dsk, keys, **kwargs):
        # We cull unnecessary tasks here. Note that this isn't necessary,
        # dask will do this automatically, this just shows one optimization
        # you could do.
        dsk2, _ = cull(dsk, keys)
        return dsk2
```

```
# Use the threaded scheduler by default.
__dask_scheduler__ = staticmethod(dask.threaded.get)

def __dask_postcompute__(self):
    # We want to return the results as a tuple, so our finalize
    # function is `tuple`. There are no extra arguments, so we also
    # return an empty tuple.
    return tuple, ()

def __dask_postpersist__(self):
    # Since our __init__ takes a graph as its first argument, our
    # rebuild function can just be the class itself. For extra
    # arguments we also return a tuple containing just the keys.
    return Tuple, (self._keys,)

def __dask_tokenize__(self):
    # For tokenize to work we want to return a value that fully
    # represents this object. In this case it's the list of keys
    # to be computed.
    return tuple(self._keys)
```

Demonstrating this class:

```
>>> from dask_tuple import Tuple
>>> from operator import add, mul

# Define a dask graph
>>> dsk = {'a': 1,
...       'b': 2,
...       'c': (add, 'a', 'b'),
...       'd': (mul, 'b', 2),
...       'e': (add, 'b', 'c')}

# The output keys for this graph
>>> keys = ['b', 'c', 'd', 'e']

>>> x = Tuple(dsk, keys)

# Compute turns Tuple into a tuple
>>> x.compute()
(2, 3, 4, 5)

# Persist turns Tuple into a Tuple, with each task already computed
>>> x2 = x.persist()
>>> isinstance(x2, Tuple)
True
>>> x2.__dask_graph__()
{'b': 2,
 'c': 3,
 'd': 4,
 'e': 5}
>>> x2.compute()
(2, 3, 4, 5)
```

### 4.32.5 Checking if an object is a dask collection

To check if an object is a dask collection, use `dask.base.is_dask_collection`:

```
>>> from dask.base import is_dask_collection
>>> from dask import delayed

>>> x = delayed(sum)([1, 2, 3])
>>> is_dask_collection(x)
True
>>> is_dask_collection(1)
False
```

### 4.32.6 Implementing Deterministic Hashing

Dask implements its own deterministic hash function to generate keys based on the value of arguments. This function is available as `dask.base.tokenize`. Many common types already have implementations of `tokenize`, which can be found in `dask/base.py`.

When creating your own custom classes you may need to register a `tokenize` implementation. There are two ways to do this:

---

**Note:** Both dask collections and normal python objects can have implementations of `tokenize` using either of the methods described below.

---

1. The `__dask_tokenize__` method

Where possible, it's recommended to define the `__dask_tokenize__` method. This method takes no arguments and should return a value fully representative of the object.

2. Register a function with `dask.base.normalize_token`

If defining a method on the class isn't possible, you can register a `tokenize` function with the `normalize_token` dispatch. The function should have the same signature as described above.

In both cases the implementation should be the same, only the location of the definition is different.

#### Example

```
>>> from dask.base import tokenize, normalize_token

# Define a tokenize implementation using a method.
>>> class Foo(object):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...
...     def __dask_tokenize__(self):
...         # This tuple fully represents self
...         return (Foo, self.a, self.b)

>>> x = Foo(1, 2)
>>> tokenize(x)
'5988362b6e07087db2bc8e7c1c8cc560'
>>> tokenize(x) == tokenize(x) # token is deterministic
```

```

True

# Register an implementation with normalize_token
>>> class Bar(object):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y

>>> @normalize_token.register(Bar)
... def tokenize_bar(x):
...     return (Bar, x.x, x.x)

>>> y = Bar(1, 2)
>>> tokenize(y)
'5a7e9c3645aa44cf13d021c14452152e'
>>> tokenize(y) == tokenize(y)
True
>>> tokenize(y) == tokenize(x) # tokens for different objects aren't equal
False

```

For more examples please see `dask/base.py` or any of the built-in dask collections.

## 4.33 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](#).

### 4.33.1 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 }
}
```

## 4.34 Funding

Dask receives generous funding and support from the following sources:

1. The time and effort of numerous [open source contributors](#)
2. [The DARPA XData program](#)
3. [The Moore Foundation's Data Driven Discovery program](#)
4. [Anaconda Inc](#)
5. A variety of private companies who sponsor the development of particular open source features

We encourage monetary donations to [NumFOCUS](#) to support open source scientific computing software.

Dask is supported by [Anaconda Inc](#) and develops under the [BSD 3-clause license](#).



---

## Bibliography

---

- [R103] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. <http://www.math.sfu.ca/~cbm/aands/>
- [R104] Wikipedia, “Inverse hyperbolic function”, <http://en.wikipedia.org/wiki/Arccosh>
- [R105] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. <http://www.math.sfu.ca/~cbm/aands/>
- [R106] Wikipedia, “Inverse hyperbolic function”, <http://en.wikipedia.org/wiki/Arcsinh>
- [R107] ISO/IEC standard 9899:1999, “Programming language C.”
- [R108] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. <http://www.math.sfu.ca/~cbm/aands/>
- [R109] Wikipedia, “Inverse hyperbolic function”, <http://en.wikipedia.org/wiki/Arctanh>
- [R110] “Lecture Notes on the Status of IEEE 754”, William Kahan, <http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>
- [R111] “How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?”, William Kahan, <http://www.cs.berkeley.edu/~wkahan/Mindless.pdf>
- [R112] Wikipedia, “Exponential function”, [http://en.wikipedia.org/wiki/Exponential\\_function](http://en.wikipedia.org/wiki/Exponential_function)
- [R113] M. Abramowitz and I. A. Stegun, “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables,” Dover, 1964, p. 69, [http://www.math.sfu.ca/~cbm/aands/page\\_69.htm](http://www.math.sfu.ca/~cbm/aands/page_69.htm)
- [R114] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. <http://www.math.sfu.ca/~cbm/aands/>
- [R115] Wikipedia, “Logarithm”. <http://en.wikipedia.org/wiki/Logarithm>
- [R116] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. <http://www.math.sfu.ca/~cbm/aands/>
- [R117] Wikipedia, “Logarithm”. <http://en.wikipedia.org/wiki/Logarithm>
- [R118] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. <http://www.math.sfu.ca/~cbm/aands/>
- [R119] Wikipedia, “Logarithm”. <http://en.wikipedia.org/wiki/Logarithm>

- [R120] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. <http://www.math.sfu.ca/~cbm/aands/>
- [R121] Wikipedia, “Hyperbolic function”, [http://en.wikipedia.org/wiki/Hyperbolic\\_function](http://en.wikipedia.org/wiki/Hyperbolic_function)
- [R122] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15
- [CT] Cooley, James W., and John W. Tukey, 1965, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.* 19: 297-301.
- [R123] Dalgaard, Peter, “Introductory Statistics with R”, Springer-Verlag, 2002.
- [R124] Glantz, Stanton A. “Primer of Biostatistics.”, McGraw-Hill, Fifth Edition, 2002.
- [R125] Lentner, Marvin, “Elementary Applied Statistics”, Bogden and Quigley, 1972.
- [R126] Weisstein, Eric W. “Binomial Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/BinomialDistribution.html>
- [R127] Wikipedia, “Binomial-distribution”, [http://en.wikipedia.org/wiki/Binomial\\_distribution](http://en.wikipedia.org/wiki/Binomial_distribution)
- [R128] NIST “Engineering Statistics Handbook” <http://www.itl.nist.gov/div898/handbook/eda/section3/eda3666.htm>
- [R129] Peyton Z. Peebles Jr., “Probability, Random Variables and Random Signal Principles”, 4th ed, 2001, p. 57.
- [R130] “Poisson Process”, Wikipedia, [http://en.wikipedia.org/wiki/Poisson\\_process](http://en.wikipedia.org/wiki/Poisson_process)
- [R131] “Exponential Distribution, Wikipedia, [http://en.wikipedia.org/wiki/Exponential\\_distribution](http://en.wikipedia.org/wiki/Exponential_distribution)
- [R132] Glantz, Stanton A. “Primer of Biostatistics.”, McGraw-Hill, Fifth Edition, 2002.
- [R133] Wikipedia, “F-distribution”, <http://en.wikipedia.org/wiki/F-distribution>
- [R134] Weisstein, Eric W. “Gamma Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/GammaDistribution.html>
- [R135] Wikipedia, “Gamma-distribution”, <http://en.wikipedia.org/wiki/Gamma-distribution>
- [R136] Gumbel, E. J., “Statistics of Extremes,” New York: Columbia University Press, 1958.
- [R137] Reiss, R.-D. and Thomas, M., “Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields,” Basel: Birkhauser Verlag, 2001.
- [R138] Lentner, Marvin, “Elementary Applied Statistics”, Bogden and Quigley, 1972.
- [R139] Weisstein, Eric W. “Hypergeometric Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/HypergeometricDistribution.html>
- [R140] Wikipedia, “Hypergeometric-distribution”, [http://en.wikipedia.org/wiki/Hypergeometric\\_distribution](http://en.wikipedia.org/wiki/Hypergeometric_distribution)
- [R141] Abramowitz, M. and Stegun, I. A. (Eds.). “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing,” New York: Dover, 1972.
- [R142] Kotz, Samuel, et. al. “The Laplace Distribution and Generalizations,” Birkhauser, 2001.
- [R143] Weisstein, Eric W. “Laplace Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/LaplaceDistribution.html>
- [R144] Wikipedia, “Laplace Distribution”, [http://en.wikipedia.org/wiki/Laplace\\_distribution](http://en.wikipedia.org/wiki/Laplace_distribution)
- [R145] 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.
- [R146] Weisstein, Eric W. “Logistic Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/LogisticDistribution.html>
- [R147] Wikipedia, “Logistic-distribution”, [http://en.wikipedia.org/wiki/Logistic\\_distribution](http://en.wikipedia.org/wiki/Logistic_distribution)

- [R148] 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>
- [R149] Reiss, R.D. and Thomas, M., “Statistical Analysis of Extreme Values,” Basel: Birkhauser Verlag, 2001, pp. 31-32.
- [R150] 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).
- [R151] 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.
- [R152] D. J. Hand, F. Daly, D. Lunn, E. Ostrowski, *A Handbook of Small Data Sets*, CRC Press, 1994.
- [R153] Wikipedia, “Logarithmic-distribution”, <http://en.wikipedia.org/wiki/Logarithmic-distribution>
- [R154] Weisstein, Eric W. “Negative Binomial Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/NegativeBinomialDistribution.html>
- [R155] Wikipedia, “Negative binomial distribution”, [http://en.wikipedia.org/wiki/Negative\\_binomial\\_distribution](http://en.wikipedia.org/wiki/Negative_binomial_distribution)
- [R156] Delhi, M.S. Holla, “On a noncentral chi-square distribution in the analysis of weapon systems effectiveness”, *Metrika*, Volume 15, Number 1 / December, 1970.
- [R157] Wikipedia, “Noncentral chi-square distribution” [http://en.wikipedia.org/wiki/Noncentral\\_chi-square\\_distribution](http://en.wikipedia.org/wiki/Noncentral_chi-square_distribution)
- [R158] Weisstein, Eric W. “Noncentral F-Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/NoncentralF-Distribution.html>
- [R159] Wikipedia, “Noncentral F distribution”, [http://en.wikipedia.org/wiki/Noncentral\\_F-distribution](http://en.wikipedia.org/wiki/Noncentral_F-distribution)
- [R160] Wikipedia, “Normal distribution”, [http://en.wikipedia.org/wiki/Normal\\_distribution](http://en.wikipedia.org/wiki/Normal_distribution)
- [R161] P. R. Peebles Jr., “Central Limit Theorem” in “Probability, Random Variables and Random Signal Principles”, 4th ed., 2001, pp. 51, 51, 125.
- [R162] Francis Hunt and Paul Johnson, *On the Pareto Distribution of Sourceforge projects*.
- [R163] Pareto, V. (1896). *Course of Political Economy*. Lausanne.
- [R164] Reiss, R.D., Thomas, M.(2001), *Statistical Analysis of Extreme Values*, Birkhauser Verlag, Basel, pp 23-30.
- [R165] Wikipedia, “Pareto distribution”, [http://en.wikipedia.org/wiki/Pareto\\_distribution](http://en.wikipedia.org/wiki/Pareto_distribution)
- [R166] Weisstein, Eric W. “Poisson Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/PoissonDistribution.html>
- [R167] Wikipedia, “Poisson distribution”, [http://en.wikipedia.org/wiki/Poisson\\_distribution](http://en.wikipedia.org/wiki/Poisson_distribution)
- [R168] Christian Kleiber, Samuel Kotz, “Statistical size distributions in economics and actuarial sciences”, Wiley, 2003.
- [R169] Heckert, N. A. and Filliben, James J. “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>
- [R170] Brighton Webs Ltd., “Rayleigh Distribution,” <http://www.brighton-webs.co.uk/distributions/rayleigh.asp>
- [R171] Wikipedia, “Rayleigh distribution” [http://en.wikipedia.org/wiki/Rayleigh\\_distribution](http://en.wikipedia.org/wiki/Rayleigh_distribution)
- [R172] NIST/SEMATECH e-Handbook of Statistical Methods, “Cauchy Distribution”, <http://www.itl.nist.gov/div898/handbook/eda/section3/eda3663.htm>

- [R173] Weisstein, Eric W. “Cauchy Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/CauchyDistribution.html>
- [R174] Wikipedia, “Cauchy distribution” [http://en.wikipedia.org/wiki/Cauchy\\_distribution](http://en.wikipedia.org/wiki/Cauchy_distribution)
- [R175] Weisstein, Eric W. “Gamma Distribution.” From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/GammaDistribution.html>
- [R176] Wikipedia, “Gamma-distribution”, <http://en.wikipedia.org/wiki/Gamma-distribution>
- [R177] Dalgaard, Peter, “Introductory Statistics With R”, Springer, 2002.
- [R178] Wikipedia, “Student’s t-distribution” [http://en.wikipedia.org/wiki/Student’s\\_t-distribution](http://en.wikipedia.org/wiki/Student’s_t-distribution)
- [R179] Wikipedia, “Triangular distribution” [http://en.wikipedia.org/wiki/Triangular\\_distribution](http://en.wikipedia.org/wiki/Triangular_distribution)
- [R180] Abramowitz, M. and Stegun, I. A. (Eds.). “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing,” New York: Dover, 1972.
- [R181] von Mises, R., “Mathematical Theory of Probability and Statistics”, New York: Academic Press, 1964.
- [R182] Brighton Webs Ltd., Wald Distribution, <http://www.brighton-webs.co.uk/distributions/wald.asp>
- [R183] Chhikara, Raj S., and Folks, J. Leroy, “The Inverse Gaussian Distribution: Theory : Methodology, and Applications”, CRC Press, 1988.
- [R184] Wikipedia, “Wald distribution” [http://en.wikipedia.org/wiki/Wald\\_distribution](http://en.wikipedia.org/wiki/Wald_distribution)
- [R185] Waloddi Weibull, Royal Technical University, Stockholm, 1939 “A Statistical Theory Of The Strength Of Materials”, Ingeniorsvetenskapsakademiens Handlingar Nr 151, 1939, Generalstabens Litografiska Anstalts Forlag, Stockholm.
- [R186] Waloddi Weibull, “A Statistical Distribution Function of Wide Applicability”, Journal Of Applied Mechanics ASME Paper 1951.
- [R187] Wikipedia, “Weibull distribution”, [http://en.wikipedia.org/wiki/Weibull\\_distribution](http://en.wikipedia.org/wiki/Weibull_distribution)
- [R188] [http://en.wikipedia.org/wiki/T-test#Independent\\_two-sample\\_t-test](http://en.wikipedia.org/wiki/T-test#Independent_two-sample_t-test)
- [R189] [http://en.wikipedia.org/wiki/Welch%27s\\_t\\_test](http://en.wikipedia.org/wiki/Welch%27s_t_test)
- [R190] Lowry, Richard. “Concepts and Applications of Inferential Statistics”. Chapter 8. <http://faculty.vassar.edu/lowry/ch8pt1.html>
- [R191] “Chi-squared test”, [http://en.wikipedia.org/wiki/Chi-squared\\_test](http://en.wikipedia.org/wiki/Chi-squared_test)
- [R192] Lowry, Richard. “Concepts and Applications of Inferential Statistics”. Chapter 8. <http://faculty.vassar.edu/lowry/ch8pt1.html>
- [R193] “Chi-squared test”, [http://en.wikipedia.org/wiki/Chi-squared\\_test](http://en.wikipedia.org/wiki/Chi-squared_test)
- [R194] “G-test”, <http://en.wikipedia.org/wiki/G-test>
- [R195] Sokal, R. R. and Rohlf, F. J. “Biometry: the principles and practice of statistics in biological research”, New York: Freeman (1981)
- [R196] Cressie, N. and Read, T. R. C., “Multinomial Goodness-of-Fit Tests”, J. Royal Stat. Soc. Series B, Vol. 46, No. 3 (1984), pp. 440-464.
- [R197] Zwillinger, D. and Kokoska, S. (2000). CRC Standard Probability and Statistics Tables and Formulae. Chapman & Hall: New York. 2000. Section 2.2.24.1
- [R198] Zwillinger, D. and Kokoska, S. (2000). CRC Standard Probability and Statistics Tables and Formulae. Chapman & Hall: New York. 2000.
- [R199] D’Agostino, R. B. (1971), “An omnibus test of normality for moderate and large sample size,” Biometrika, 58, 341-348

- [R200] D'Agostino, R. and Pearson, E. S. (1973), "Testing for departures from normality," *Biometrika*, 60, 613-622
- [R201] Lowry, Richard. "Concepts and Applications of Inferential Statistics". Chapter 14. <http://faculty.vassar.edu/lowry/ch14pt1.html>
- [R202] Heiman, G.W. *Research Methods in Statistics*. 2002.
- [R203] McDonald, G. H. "Handbook of Biological Statistics", One-way ANOVA. <http://http://www.biostathandbook.com/onewayanova.html>
- [R204] <http://eli.thegreenplace.net/2009/03/21/efficient-integer-exponentiation-algorithms>
- [R205] Pebay, Philippe (2008), "Formulas for Robust, One-Pass Parallel



## Symbols

- `__dask_graph__()`, 605
  - `__dask_keys__()`, 606
  - `__dask_optimize__()`, 606
  - `__dask_postcompute__()`, 606
  - `__dask_postpersist__()`, 607
  - `__dask_scheduler__()`, 606
- ## A
- `abs()` (dask.dataframe.DataFrame method), 328
  - `abs()` (dask.dataframe.Series method), 403
  - `accumulate()` (dask.bag.Bag method), 299
  - `add()` (dask.dataframe.DataFrame method), 328
  - `add()` (dask.dataframe.Series method), 403
  - `add_done_callback()` (distributed.Future method), 536
  - `agg()` (dask.dataframe.groupby.DataFrameGroupBy method), 466
  - `agg()` (dask.dataframe.groupby.SeriesGroupBy method), 472
  - `aggregate()` (dask.dataframe.groupby.DataFrameGroupBy method), 467
  - `aggregate()` (dask.dataframe.groupby.SeriesGroupBy method), 473
  - `align()` (dask.dataframe.DataFrame method), 328
  - `align()` (dask.dataframe.Series method), 403
  - `all()` (dask.array.Array method), 274
  - `all()` (dask.bag.Bag method), 300
  - `all()` (dask.dataframe.DataFrame method), 329
  - `all()` (dask.dataframe.Series method), 404
  - `all()` (in module dask.array), 60
  - `allclose()` (in module dask.array), 62
  - `angle()` (in module dask.array), 63
  - `any()` (dask.array.Array method), 274
  - `any()` (dask.bag.Bag method), 300
  - `any()` (dask.dataframe.DataFrame method), 329
  - `any()` (dask.dataframe.Series method), 405
  - `any()` (in module dask.array), 63
  - `append()` (dask.dataframe.DataFrame method), 330
  - `append()` (dask.dataframe.Series method), 405
  - `apply()` (dask.dataframe.DataFrame method), 331
  - `apply()` (dask.dataframe.groupby.DataFrameGroupBy method), 468
  - `apply()` (dask.dataframe.groupby.SeriesGroupBy method), 474
  - `apply()` (dask.dataframe.Series method), 406
  - `apply_along_axis()` (in module dask.array), 64
  - `apply_over_axes()` (in module dask.array), 65
  - `applymap()` (dask.dataframe.DataFrame method), 332
  - `arange()` (in module dask.array), 66
  - `arccos()` (in module dask.array), 67
  - `arccosh()` (in module dask.array), 68
  - `arcsin()` (in module dask.array), 68
  - `arcsinh()` (in module dask.array), 69
  - `arctan()` (in module dask.array), 70
  - `arctan2()` (in module dask.array), 71
  - `arctanh()` (in module dask.array), 72
  - `argmax()` (dask.array.Array method), 274
  - `argmax()` (in module dask.array), 73
  - `argmin()` (dask.array.Array method), 274
  - `argmin()` (in module dask.array), 74
  - `argwhere()` (in module dask.array), 75
  - `around()` (in module dask.array), 75
  - `Array` (class in dask.array), 273
  - `array()` (in module dask.array), 76
  - `as_completed()` (in module distributed), 516
  - `asanyarray()` (in module dask.array), 78
  - `asarray()` (in module dask.array), 78
  - `assign()` (dask.dataframe.DataFrame method), 333
  - `astype()` (dask.array.Array method), 274
  - `astype()` (dask.dataframe.DataFrame method), 334
  - `astype()` (dask.dataframe.Series method), 407
  - `asynchronous` (distributed.Client attribute), 519
  - `atleast_1d()` (in module dask.array), 78
  - `atleast_2d()` (in module dask.array), 79
  - `atleast_3d()` (in module dask.array), 80
  - `atop()` (in module dask.array.core), 270
  - `autocorr()` (dask.dataframe.Series method), 409

## B

Bag (class in `dask.bag`), 299  
`beta()` (in module `dask.array.random`), 228  
`between()` (`dask.dataframe.Series` method), 409  
`bfill()` (`dask.dataframe.DataFrame` method), 335  
`bfill()` (`dask.dataframe.Series` method), 409  
`bincount()` (in module `dask.array`), 80  
`binomial()` (in module `dask.array.random`), 228  
`block()` (in module `dask.array`), 81  
`broadcast_to()` (in module `dask.array`), 83

## C

`call_stack()` (`distributed.Client` method), 519  
`cancel()` (`distributed.Client` method), 519  
`cancel()` (`distributed.Future` method), 536  
`cancelled()` (`distributed.Future` method), 536  
`categorize()` (`dask.dataframe.DataFrame` method), 336  
`ceil()` (in module `dask.array`), 84  
`chisquare()` (in module `dask.array.random`), 229  
`chisquare()` (in module `dask.array.stats`), 259  
`cholesky()` (in module `dask.array.linalg`), 186  
`choose()` (`dask.array.Array` method), 275  
`choose()` (in module `dask.array`), 84  
`clear_divisions()` (`dask.dataframe.DataFrame` method), 336  
`clear_divisions()` (`dask.dataframe.Series` method), 409  
`Client` (class in `distributed`), 518  
`clip()` (`dask.array.Array` method), 275  
`clip()` (`dask.dataframe.DataFrame` method), 336  
`clip()` (`dask.dataframe.Series` method), 409  
`clip()` (in module `dask.array`), 86  
`clip_lower()` (`dask.dataframe.DataFrame` method), 337  
`clip_lower()` (`dask.dataframe.Series` method), 410  
`clip_upper()` (`dask.dataframe.DataFrame` method), 338  
`clip_upper()` (`dask.dataframe.Series` method), 411  
`close()` (`distributed.Client` method), 519  
`close()` (`distributed.deploy.local.LocalCluster` method), 14  
`coarsen()` (in module `dask.array`), 58, 83  
`combine()` (`dask.dataframe.DataFrame` method), 338  
`combine()` (`dask.dataframe.Series` method), 411  
`combine_first()` (`dask.dataframe.DataFrame` method), 339  
`combine_first()` (`dask.dataframe.Series` method), 412  
`compress()` (in module `dask.array`), 87  
`compute()` (`dask.dataframe.DataFrame` method), 339  
`compute()` (`dask.dataframe.Series` method), 412  
`compute()` (`distributed.Client` method), 520  
`compute()` (in module `dask.dataframe`), 491  
`concat()` (in module `dask.bag`), 313  
`concat()` (in module `dask.dataframe.multi`), 492  
`concatenate()` (in module `dask.array`), 60, 88  
`conj()` (in module `dask.array`), 89  
`copy()` (`dask.array.Array` method), 275  
`copy()` (`dask.dataframe.DataFrame` method), 339  
`copy()` (`dask.dataframe.Series` method), 412

`copysign()` (in module `dask.array`), 89  
`corr()` (`dask.dataframe.DataFrame` method), 339  
`corr()` (`dask.dataframe.Series` method), 413  
`corrcoef()` (in module `dask.array`), 90  
`cos()` (in module `dask.array`), 90  
`cosh()` (in module `dask.array`), 91  
`count()` (`dask.bag.Bag` method), 300  
`count()` (`dask.dataframe.DataFrame` method), 340  
`count()` (`dask.dataframe.groupby.DataFrameGroupBy` method), 469  
`count()` (`dask.dataframe.groupby.SeriesGroupBy` method), 474  
`count()` (`dask.dataframe.Series` method), 413  
`count_nonzero()` (in module `dask.array`), 92  
`cov()` (`dask.dataframe.DataFrame` method), 340  
`cov()` (`dask.dataframe.Series` method), 413  
`cov()` (in module `dask.array`), 92  
`cumcount()` (`dask.dataframe.groupby.DataFrameGroupBy` method), 469  
`cumcount()` (`dask.dataframe.groupby.SeriesGroupBy` method), 474  
`cummax()` (`dask.dataframe.DataFrame` method), 340  
`cummax()` (`dask.dataframe.Series` method), 413  
`cummin()` (`dask.dataframe.DataFrame` method), 341  
`cummin()` (`dask.dataframe.Series` method), 414  
`cumprod()` (`dask.array.Array` method), 275  
`cumprod()` (`dask.dataframe.DataFrame` method), 341  
`cumprod()` (`dask.dataframe.groupby.DataFrameGroupBy` method), 470  
`cumprod()` (`dask.dataframe.groupby.SeriesGroupBy` method), 475  
`cumprod()` (`dask.dataframe.Series` method), 414  
`cumprod()` (in module `dask.array`), 94  
`cumsum()` (`dask.array.Array` method), 275  
`cumsum()` (`dask.dataframe.DataFrame` method), 341  
`cumsum()` (`dask.dataframe.groupby.DataFrameGroupBy` method), 470  
`cumsum()` (`dask.dataframe.groupby.SeriesGroupBy` method), 475  
`cumsum()` (`dask.dataframe.Series` method), 414  
`cumsum()` (in module `dask.array`), 95  
`current()` (`distributed.Client` class method), 521

## D

`DataFrame` (class in `dask.dataframe`), 327  
`DataFrameGroupBy` (class in `dask.dataframe.groupby`), 466  
`deg2rad()` (in module `dask.array`), 96  
`degrees()` (in module `dask.array`), 96  
`delayed()` (in module `dask.delayed`), 505  
`delete()` (`distributed.Variable` method), 538  
`describe()` (`dask.dataframe.DataFrame` method), 341  
`describe()` (`dask.dataframe.Series` method), 414  
`diag()` (in module `dask.array`), 97

diff() (dask.dataframe.DataFrame method), 345  
 diff() (dask.dataframe.Series method), 418  
 diff() (in module dask.array), 98  
 digitize() (in module dask.array), 98  
 distinct() (dask.bag.Bag method), 300  
 div() (dask.dataframe.DataFrame method), 345  
 div() (dask.dataframe.Series method), 418  
 done() (distributed.Future method), 536  
 dot() (dask.array.Array method), 275  
 dot() (in module dask.array), 100  
 drop() (dask.dataframe.DataFrame method), 346  
 drop\_duplicates() (dask.dataframe.DataFrame method), 347  
 drop\_duplicates() (dask.dataframe.Series method), 418  
 dropna() (dask.dataframe.DataFrame method), 347  
 dropna() (dask.dataframe.Series method), 419  
 dstack() (in module dask.array), 101  
 dtype (dask.dataframe.Series attribute), 419  
 dtypes (dask.dataframe.DataFrame attribute), 349

## E

ediff1d() (in module dask.array), 102  
 empty() (in module dask.array), 103  
 empty\_like() (in module dask.array), 103  
 eq() (dask.dataframe.DataFrame method), 349  
 eq() (dask.dataframe.Series method), 419  
 eval() (dask.dataframe.DataFrame method), 349  
 exception() (distributed.Future method), 536  
 exp() (in module dask.array), 104  
 expm1() (in module dask.array), 105  
 exponential() (in module dask.array.random), 230  
 eye() (in module dask.array), 105

## F

f() (in module dask.array.random), 231  
 f\_oneway() (in module dask.array.stats), 266  
 fabs() (in module dask.array), 106  
 ffill() (dask.dataframe.DataFrame method), 349  
 ffill() (dask.dataframe.Series method), 419  
 fft() (in module dask.array.fft), 207  
 fft2() (in module dask.array.fft), 209  
 fft\_wrap() (in module dask.array.fft), 207  
 fftfreq() (in module dask.array.fft), 225  
 fftn() (in module dask.array.fft), 210  
 fftshift() (in module dask.array.fft), 227  
 filled() (in module dask.array.ma), 191  
 fillna() (dask.dataframe.DataFrame method), 350  
 fillna() (dask.dataframe.Series method), 420  
 filter() (dask.bag.Bag method), 300  
 fire\_and\_forget() (in module distributed), 516  
 first() (dask.dataframe.DataFrame method), 351  
 first() (dask.dataframe.Series method), 421  
 fix() (in module dask.array), 106  
 fix\_invalid() (in module dask.array.ma), 192

flatnonzero() (in module dask.array), 107  
 flatten() (dask.array.Array method), 275  
 flatten() (dask.bag.Bag method), 300  
 flip() (in module dask.array), 107  
 fliplr() (in module dask.array), 108  
 flipud() (in module dask.array), 108  
 floor() (in module dask.array), 109  
 floordiv() (dask.dataframe.DataFrame method), 351  
 floordiv() (dask.dataframe.Series method), 422  
 fmax() (in module dask.array), 109  
 fmin() (in module dask.array), 110  
 fmod() (in module dask.array), 111  
 fold() (dask.bag.Bag method), 300  
 foldby() (dask.bag.Bag method), 301  
 frequencies() (dask.bag.Bag method), 302  
 frexp() (in module dask.array), 112  
 from\_array() (in module dask.array), 56, 204  
 from\_array() (in module dask.dataframe), 482  
 from\_bcolz() (in module dask.dataframe), 483  
 from\_dask\_array() (in module dask.dataframe), 484  
 from\_delayed() (in module dask.array), 57, 205  
 from\_delayed() (in module dask.bag), 311  
 from\_delayed() (in module dask.dataframe), 484  
 from\_npy\_stack() (in module dask.array), 205  
 from\_pandas() (in module dask.dataframe), 482  
 from\_sequence() (in module dask.bag), 310  
 from\_url() (in module dask.bag), 312  
 fromfunction() (in module dask.array), 113  
 frompyfunc() (in module dask.array), 113  
 full() (in module dask.array), 114  
 full\_like() (in module dask.array), 115  
 Future (class in distributed), 536

## G

gamma() (in module dask.array.random), 232  
 gather() (distributed.Client method), 521  
 ge() (dask.dataframe.DataFrame method), 352  
 ge() (dask.dataframe.Series method), 422  
 geometric() (in module dask.array.random), 233  
 get() (distributed.Client method), 521  
 get() (distributed.Queue method), 537  
 get() (distributed.Variable method), 538  
 get\_client() (in module distributed), 517  
 get\_dataset() (distributed.Client method), 522  
 get\_dtype\_counts() (dask.dataframe.DataFrame method), 352  
 get\_executor() (distributed.Client method), 522  
 get\_fstype\_counts() (dask.dataframe.DataFrame method), 352  
 get\_group() (dask.dataframe.groupby.DataFrameGroupBy method), 470  
 get\_group() (dask.dataframe.groupby.SeriesGroupBy method), 475  
 get\_metadata() (distributed.Client method), 522

get\_partition() (dask.dataframe.DataFrame method), 352  
 get\_partition() (dask.dataframe.Series method), 422  
 get\_restrictions() (distributed.Client class method), 522  
 get\_scheduler\_logs() (distributed.Client method), 522  
 get\_versions() (distributed.Client method), 522  
 get\_worker\_logs() (distributed.Client method), 523  
 getdata() (in module dask.array.ma), 193  
 getmaskarray() (in module dask.array.ma), 193  
 ghost() (in module dask.array.ghost), 203  
 glob() (dask.bytes.local.LocalFileSystem method), 604  
 groupby() (dask.bag.Bag method), 302  
 groupby() (dask.dataframe.DataFrame method), 352  
 groupby() (dask.dataframe.Series method), 422  
 gt() (dask.dataframe.DataFrame method), 353  
 gt() (dask.dataframe.Series method), 423  
 gumbel() (in module dask.array.random), 233

## H

has\_what() (distributed.Client method), 523  
 head() (dask.dataframe.DataFrame method), 353  
 head() (dask.dataframe.Series method), 424  
 hfft() (in module dask.array.fft), 223  
 histogram() (in module dask.array), 116  
 hstack() (in module dask.array), 116  
 hypergeometric() (in module dask.array.random), 235  
 hypot() (in module dask.array), 117

## I

idxmax() (dask.dataframe.DataFrame method), 354  
 idxmax() (dask.dataframe.Series method), 424  
 idxmin() (dask.dataframe.DataFrame method), 354  
 idxmin() (dask.dataframe.Series method), 424  
 ifft() (in module dask.array.fft), 212  
 ifft2() (in module dask.array.fft), 213  
 ifftn() (in module dask.array.fft), 215  
 ifftshift() (in module dask.array.fft), 227  
 ihfft() (in module dask.array.fft), 224  
 imag() (in module dask.array), 117  
 imread() (in module dask.array.image), 267  
 index (dask.dataframe.DataFrame attribute), 354  
 index (dask.dataframe.Series attribute), 425  
 indices() (in module dask.array), 118  
 info() (dask.dataframe.DataFrame method), 354  
 insert() (in module dask.array), 118  
 inv() (in module dask.array.linalg), 186  
 irfft() (in module dask.array.fft), 220  
 irfft2() (in module dask.array.fft), 221  
 irfftn() (in module dask.array.fft), 222  
 isclose() (in module dask.array), 120  
 iscomplex() (in module dask.array), 121  
 isfinite() (in module dask.array), 121  
 isin() (dask.dataframe.DataFrame method), 354  
 isin() (dask.dataframe.Series method), 425  
 isinf() (in module dask.array), 122

isnan() (in module dask.array), 123  
 isnull() (dask.dataframe.DataFrame method), 355  
 isnull() (dask.dataframe.Series method), 425  
 isnull() (in module dask.array), 123  
 isreal() (in module dask.array), 124  
 itemsize (dask.array.Array attribute), 276  
 iteritems() (dask.dataframe.Series method), 426  
 iterrows() (dask.dataframe.DataFrame method), 355  
 itertuples() (dask.dataframe.DataFrame method), 356

## J

join() (dask.bag.Bag method), 303  
 join() (dask.dataframe.DataFrame method), 357

## K

known\_divisions (dask.dataframe.DataFrame attribute), 359  
 known\_divisions (dask.dataframe.Series attribute), 426  
 kurtosis() (in module dask.array.stats), 264  
 kurtosistest() (in module dask.array.stats), 264

## L

laplace() (in module dask.array.random), 236  
 last() (dask.dataframe.DataFrame method), 359  
 last() (dask.dataframe.Series method), 426  
 ldexp() (in module dask.array), 124  
 le() (dask.dataframe.DataFrame method), 359  
 le() (dask.dataframe.Series method), 426  
 linspace() (in module dask.array), 125  
 list\_datasets() (distributed.Client method), 523  
 loc (dask.dataframe.DataFrame attribute), 359  
 loc (dask.dataframe.Series attribute), 426  
 LocalCluster (class in distributed.deploy.local), 13  
 LocalFileSystem (class in dask.bytes.local), 604  
 log() (in module dask.array), 125  
 log10() (in module dask.array), 126  
 log1p() (in module dask.array), 126  
 log2() (in module dask.array), 127  
 logaddexp() (in module dask.array), 128  
 logaddexp2() (in module dask.array), 128  
 logical\_and() (in module dask.array), 129  
 logical\_not() (in module dask.array), 130  
 logical\_or() (in module dask.array), 130  
 logical\_xor() (in module dask.array), 131  
 logistic() (in module dask.array.random), 237  
 lognormal() (in module dask.array.random), 238  
 logseries() (in module dask.array.random), 239  
 lstsq() (in module dask.array.linalg), 186  
 lt() (dask.dataframe.DataFrame method), 359  
 lt() (dask.dataframe.Series method), 426  
 lu() (in module dask.array.linalg), 187

## M

map() (dask.bag.Bag method), 303

- map() (dask.dataframe.Series method), 427
- map() (distributed.Client method), 523
- map() (in module dask.bag), 313
- map\_blocks() (dask.array.Array method), 276
- map\_blocks() (in module dask.array.core), 268
- map\_overlap() (dask.array.Array method), 278
- map\_overlap() (dask.dataframe.DataFrame method), 359
- map\_overlap() (dask.dataframe.Series method), 428
- map\_overlap() (in module dask.array.ghost), 204
- map\_overlap() (in module dask.dataframe.rolling), 490
- map\_partitions() (dask.bag.Bag method), 304
- map\_partitions() (dask.dataframe.DataFrame method), 361
- map\_partitions() (dask.dataframe.Series method), 430
- map\_partitions() (in module dask.bag), 313
- map\_partitions() (in module dask.dataframe), 491
- mask() (dask.dataframe.DataFrame method), 362
- mask() (dask.dataframe.Series method), 431
- masked\_array() (in module dask.array.ma), 194
- masked\_equal() (in module dask.array.ma), 195
- masked\_greater() (in module dask.array.ma), 196
- masked\_greater\_equal() (in module dask.array.ma), 196
- masked\_inside() (in module dask.array.ma), 196
- masked\_invalid() (in module dask.array.ma), 197
- masked\_less() (in module dask.array.ma), 197
- masked\_less\_equal() (in module dask.array.ma), 198
- masked\_not\_equal() (in module dask.array.ma), 198
- masked\_outside() (in module dask.array.ma), 199
- masked\_values() (in module dask.array.ma), 199
- masked\_where() (in module dask.array.ma), 200
- matmul() (in module dask.array), 131
- max() (dask.array.Array method), 279
- max() (dask.bag.Bag method), 305
- max() (dask.dataframe.DataFrame method), 364
- max() (dask.dataframe.groupby.DataFrameGroupBy method), 471
- max() (dask.dataframe.groupby.SeriesGroupBy method), 476
- max() (dask.dataframe.Series method), 433
- max() (in module dask.array), 133
- maximum() (in module dask.array), 134
- mean() (dask.array.Array method), 279
- mean() (dask.bag.Bag method), 305
- mean() (dask.dataframe.DataFrame method), 365
- mean() (dask.dataframe.groupby.DataFrameGroupBy method), 471
- mean() (dask.dataframe.groupby.SeriesGroupBy method), 476
- mean() (dask.dataframe.Series method), 434
- mean() (in module dask.array), 135
- memory\_usage() (dask.dataframe.DataFrame method), 365
- memory\_usage() (dask.dataframe.Series method), 434
- merge() (dask.dataframe.DataFrame method), 366
- merge() (in module dask.dataframe.multi), 493
- meshgrid() (in module dask.array), 136
- min() (dask.array.Array method), 279
- min() (dask.bag.Bag method), 305
- min() (dask.dataframe.DataFrame method), 368
- min() (dask.dataframe.groupby.DataFrameGroupBy method), 471
- min() (dask.dataframe.groupby.SeriesGroupBy method), 476
- min() (dask.dataframe.Series method), 435
- min() (in module dask.array), 138
- minimum() (in module dask.array), 139
- makedirs() (dask.bytes.local.LocalFileSystem method), 604
- mod() (dask.dataframe.DataFrame method), 368
- mod() (dask.dataframe.Series method), 435
- modf() (in module dask.array), 140
- moment() (dask.array.Array method), 279
- moment() (in module dask.array), 140
- moment() (in module dask.array.stats), 266
- mul() (dask.dataframe.DataFrame method), 369
- mul() (dask.dataframe.Series method), 435

## N

- nanargmax() (in module dask.array), 141
- nanargmin() (in module dask.array), 141
- nancumprod() (in module dask.array), 141
- nancumsum() (in module dask.array), 142
- nanmax() (in module dask.array), 143
- nanmean() (in module dask.array), 144
- nanmin() (in module dask.array), 145
- nanprod() (in module dask.array), 146
- nanstd() (in module dask.array), 147
- nansum() (in module dask.array), 148
- nanvar() (in module dask.array), 150
- nbytes (dask.array.Array attribute), 280
- nbytes (dask.dataframe.Series attribute), 436
- nbytes() (distributed.Client method), 524
- ncores() (distributed.Client method), 524
- ndim (dask.dataframe.DataFrame attribute), 369
- ndim (dask.dataframe.Series attribute), 436
- ne() (dask.dataframe.DataFrame method), 369
- ne() (dask.dataframe.Series method), 436
- negative\_binomial() (in module dask.array.random), 240
- nextafter() (in module dask.array), 151
- nlargest() (dask.dataframe.DataFrame method), 369
- nlargest() (dask.dataframe.Series method), 436
- noncentral\_chisquare() (in module dask.array.random), 241
- noncentral\_f() (in module dask.array.random), 242
- nonzero() (dask.array.Array method), 280
- nonzero() (in module dask.array), 152
- norm() (in module dask.array.linalg), 187
- normal() (in module dask.array.random), 242
- normalize\_collection() (distributed.Client method), 525

normaltest() (in module dask.array.stats), 265  
notnull() (dask.dataframe.DataFrame method), 370  
notnull() (dask.dataframe.Series method), 437  
notnull() (in module dask.array), 153  
npartitions (dask.dataframe.DataFrame attribute), 370  
npartitions (dask.dataframe.Series attribute), 437  
nsmallest() (dask.dataframe.DataFrame method), 370  
nsmallest() (dask.dataframe.Series method), 437  
nunique() (dask.dataframe.Series method), 438  
nunique\_approx() (dask.dataframe.DataFrame method), 371  
nunique\_approx() (dask.dataframe.Series method), 438

## O

ones() (in module dask.array), 153  
ones\_like() (in module dask.array), 154  
open() (dask.bytes.local.LocalFileSystem method), 604  
open\_files() (in module dask.bytes), 601

## P

pareto() (in module dask.array.random), 244  
percentile() (in module dask.array), 154  
persist() (dask.dataframe.DataFrame method), 371  
persist() (dask.dataframe.Series method), 438  
persist() (distributed.Client method), 525  
pipe() (dask.dataframe.DataFrame method), 371  
pipe() (dask.dataframe.Series method), 439  
pivot\_table() (dask.dataframe.DataFrame method), 372  
pluck() (dask.bag.Bag method), 305  
poisson() (in module dask.array.random), 245  
pow() (dask.dataframe.DataFrame method), 372  
pow() (dask.dataframe.Series method), 440  
power() (in module dask.array.random), 245  
power\_divergence() (in module dask.array.stats), 261  
processing() (distributed.Client method), 526  
prod() (dask.array.Array method), 280  
prod() (dask.dataframe.DataFrame method), 373  
prod() (dask.dataframe.Series method), 440  
prod() (in module dask.array), 154  
product() (dask.bag.Bag method), 305  
profile() (distributed.Client method), 526  
ptp() (in module dask.array), 156  
publish\_dataset() (distributed.Client method), 527  
put() (distributed.Queue method), 538

## Q

qr() (in module dask.array.linalg), 189  
qsize() (distributed.Queue method), 538  
quantile() (dask.dataframe.DataFrame method), 374  
quantile() (dask.dataframe.Series method), 441  
query() (dask.dataframe.DataFrame method), 374  
Queue (class in distributed), 537

## R

rad2deg() (in module dask.array), 156  
radd() (dask.dataframe.DataFrame method), 374  
radd() (dask.dataframe.Series method), 441  
radians() (in module dask.array), 157  
random() (in module dask.array.random), 246  
random\_sample() (dask.bag.Bag method), 305  
random\_sample() (in module dask.array.random), 247  
random\_split() (dask.dataframe.DataFrame method), 375  
random\_split() (dask.dataframe.Series method), 441  
range() (in module dask.bag), 312  
ravel() (dask.array.Array method), 280  
ravel() (in module dask.array), 157  
rayleigh() (in module dask.array.random), 247  
rdiv() (dask.dataframe.DataFrame method), 375  
rdiv() (dask.dataframe.Series method), 442  
read\_bytes() (in module dask.bytes), 600  
read\_csv() (in module dask.dataframe), 477  
read\_hdf() (in module dask.dataframe), 480  
read\_parquet() (in module dask.dataframe), 479  
read\_sql\_table() (in module dask.dataframe), 481  
read\_table() (in module dask.dataframe), 478  
read\_text() (in module dask.bag), 311  
real() (in module dask.array), 159  
rebalance() (distributed.Client method), 527  
rechunk() (dask.array.Array method), 280  
rechunk() (in module dask.array), 159  
reduction() (dask.bag.Bag method), 306  
reduction() (dask.dataframe.DataFrame method), 376  
reduction() (dask.dataframe.Series method), 442  
remove() (dask.bag.Bag method), 306  
rename() (dask.dataframe.DataFrame method), 378  
rename() (dask.dataframe.Series method), 444  
repartition() (dask.bag.Bag method), 306  
repartition() (dask.dataframe.DataFrame method), 379  
repartition() (dask.dataframe.Series method), 444  
repeat() (dask.array.Array method), 280  
repeat() (in module dask.array), 160  
replicate() (distributed.Client method), 527  
resample() (dask.dataframe.DataFrame method), 380  
resample() (dask.dataframe.Series method), 445  
reset\_index() (dask.dataframe.DataFrame method), 383  
reset\_index() (dask.dataframe.Series method), 449  
reshape() (dask.array.Array method), 280  
reshape() (in module dask.array), 161  
restart() (distributed.Client method), 528  
result() (distributed.Future method), 537  
result\_type() (in module dask.array), 161  
RewriteRule() (in module dask.rewrite), 564  
rfft() (in module dask.array.fft), 216  
rfft2() (in module dask.array.fft), 218  
rfftfreq() (in module dask.array.fft), 226  
rfftn() (in module dask.array.fft), 218  
rfloordiv() (dask.dataframe.DataFrame method), 383

- rfloordiv() (dask.dataframe.Series method), 449  
 rint() (in module dask.array), 162  
 rmod() (dask.dataframe.DataFrame method), 384  
 rmod() (dask.dataframe.Series method), 449  
 rmul() (dask.dataframe.DataFrame method), 384  
 rmul() (dask.dataframe.Series method), 449  
 roll() (in module dask.array), 162  
 rolling() (dask.dataframe.DataFrame method), 385  
 rolling() (dask.dataframe.Series method), 450  
 round() (dask.array.Array method), 280  
 round() (dask.dataframe.DataFrame method), 385  
 round() (dask.dataframe.Series method), 450  
 round() (in module dask.array), 163  
 rpow() (dask.dataframe.DataFrame method), 386  
 rpow() (dask.dataframe.Series method), 451  
 rsub() (dask.dataframe.DataFrame method), 387  
 rsub() (dask.dataframe.Series method), 451  
 rtruediv() (dask.dataframe.DataFrame method), 387  
 rtruediv() (dask.dataframe.Series method), 451  
 RuleSet() (in module dask.rewrite), 565  
 run() (distributed.Client method), 528  
 run\_coroutine() (distributed.Client method), 529  
 run\_on\_scheduler() (distributed.Client method), 529
- ## S
- sample() (dask.dataframe.DataFrame method), 388  
 sample() (dask.dataframe.Series method), 452  
 scale\_down() (distributed.deploy.local.LocalCluster method), 14  
 scale\_up() (distributed.deploy.local.LocalCluster method), 14  
 scatter() (distributed.Client method), 530  
 scheduler\_info() (distributed.Client method), 531  
 secede() (in module distributed), 517  
 select\_dtypes() (dask.dataframe.DataFrame method), 388  
 sem() (dask.dataframe.DataFrame method), 389  
 sem() (dask.dataframe.Series method), 452  
 Series (class in dask.dataframe), 403  
 SeriesGroupBy (class in dask.dataframe.groupby), 472  
 set() (distributed.Variable method), 538  
 set\_fill\_value() (in module dask.array.ma), 202  
 set\_index() (dask.dataframe.DataFrame method), 390  
 set\_metadata() (distributed.Client method), 531  
 shift() (dask.dataframe.DataFrame method), 391  
 shift() (dask.dataframe.Series method), 452  
 shutdown() (distributed.Client method), 532  
 sign() (in module dask.array), 163  
 signbit() (in module dask.array), 164  
 sin() (in module dask.array), 164  
 sinh() (in module dask.array), 165  
 size (dask.array.Array attribute), 281  
 size (dask.dataframe.DataFrame attribute), 391  
 size (dask.dataframe.Series attribute), 453  
 size() (dask.bytes.local.LocalFileSystem method), 605  
 size() (dask.dataframe.groupby.DataFrameGroupBy method), 471  
 size() (dask.dataframe.groupby.SeriesGroupBy method), 476  
 skew() (in module dask.array.stats), 263  
 skewtest() (in module dask.array.stats), 263  
 solve() (in module dask.array.linalg), 189  
 solve\_triangular() (in module dask.array.linalg), 190  
 sqrt() (in module dask.array), 166  
 square() (in module dask.array), 166  
 squeeze() (dask.array.Array method), 281  
 squeeze() (in module dask.array), 167  
 stack() (in module dask.array), 59, 167  
 stacks() (distributed.Client method), 532  
 standard\_cauchy() (in module dask.array.random), 248  
 standard\_exponential() (in module dask.array.random), 249  
 standard\_gamma() (in module dask.array.random), 249  
 standard\_normal() (in module dask.array.random), 250  
 standard\_t() (in module dask.array.random), 251  
 starmap() (dask.bag.Bag method), 306  
 start() (distributed.Client method), 532  
 start\_ipython\_scheduler() (distributed.Client method), 533  
 start\_ipython\_workers() (distributed.Client method), 533  
 start\_worker() (distributed.deploy.local.LocalCluster method), 14  
 std() (dask.array.Array method), 281  
 std() (dask.bag.Bag method), 307  
 std() (dask.dataframe.DataFrame method), 391  
 std() (dask.dataframe.groupby.DataFrameGroupBy method), 471  
 std() (dask.dataframe.groupby.SeriesGroupBy method), 476  
 std() (dask.dataframe.Series method), 453  
 std() (in module dask.array), 168  
 stop\_worker() (distributed.deploy.local.LocalCluster method), 14  
 store() (dask.array.Array method), 281  
 store() (in module dask.array), 57, 205  
 str (dask.bag.Bag attribute), 307  
 sub() (dask.dataframe.DataFrame method), 392  
 sub() (dask.dataframe.Series method), 453  
 submit() (distributed.Client method), 534  
 sum() (dask.array.Array method), 282  
 sum() (dask.bag.Bag method), 308  
 sum() (dask.dataframe.DataFrame method), 392  
 sum() (dask.dataframe.groupby.DataFrameGroupBy method), 471  
 sum() (dask.dataframe.groupby.SeriesGroupBy method), 476  
 sum() (dask.dataframe.Series method), 454  
 sum() (in module dask.array), 169  
 svd() (in module dask.array.linalg), 190

svd\_compressed() (in module dask.array.linalg), 190  
swapaxes() (dask.array.Array method), 282

## T

tail() (dask.dataframe.DataFrame method), 393  
tail() (dask.dataframe.Series method), 455  
take() (dask.bag.Bag method), 308  
take() (in module dask.array), 171  
tan() (in module dask.array), 172  
tanh() (in module dask.array), 173  
tensordot() (in module dask.array), 173  
tile() (in module dask.array), 175  
to\_bag() (dask.dataframe.DataFrame method), 393  
to\_bag() (dask.dataframe.Series method), 455  
to\_bag() (in module dask.dataframe), 488  
to\_csv() (dask.dataframe.DataFrame method), 393  
to\_csv() (dask.dataframe.Series method), 455  
to\_csv() (in module dask.dataframe), 485  
to\_dask\_dataframe() (dask.array.Array method), 282  
to\_dataframe() (dask.bag.Bag method), 308  
to\_dataframe() (dask.bag.core.Bag method), 297  
to\_delayed() (dask.array.Array method), 282  
to\_delayed() (dask.bag.Bag method), 309  
to\_delayed() (dask.bag.core.Bag method), 297  
to\_delayed() (dask.dataframe.DataFrame method), 396  
to\_delayed() (dask.dataframe.Series method), 457  
to\_delayed() (in module dask.dataframe), 485  
to\_frame() (dask.dataframe.Series method), 457  
to\_hdf() (dask.dataframe.DataFrame method), 396  
to\_hdf() (dask.dataframe.Series method), 457  
to\_hdf() (in module dask.dataframe), 488  
to\_hdf5() (dask.array.Array method), 282  
to\_hdf5() (in module dask.array), 206  
to\_html() (dask.dataframe.DataFrame method), 397  
to\_numpy\_stack() (in module dask.array), 207  
to\_parquet() (dask.dataframe.Series method), 459  
to\_parquet() (in module dask.dataframe), 489  
to\_records() (in module dask.dataframe), 485  
to\_string() (dask.dataframe.Series method), 460  
to\_textfiles() (dask.bag.Bag method), 309  
to\_textfiles() (in module dask.bag.core), 296  
to\_timestamp() (dask.dataframe.DataFrame method), 398  
to\_timestamp() (dask.dataframe.Series method), 461  
top() (in module dask.array.core), 271  
topk() (dask.array.Array method), 283  
topk() (dask.bag.Bag method), 310  
topk() (in module dask.array), 58, 176  
traceback() (distributed.Future method), 537  
transpose() (dask.array.Array method), 283  
transpose() (in module dask.array), 177  
triangular() (in module dask.array.random), 252  
tril() (in module dask.array), 177  
triu() (in module dask.array), 178  
truediv() (dask.dataframe.DataFrame method), 399

truediv() (dask.dataframe.Series method), 461  
trunc() (in module dask.array), 178  
tsqr() (in module dask.array.linalg), 191  
ttest\_1samp() (in module dask.array.stats), 258  
ttest\_ind() (in module dask.array.stats), 256  
ttest\_rel() (in module dask.array.stats), 258

## U

ukey() (dask.bytes.local.LocalFileSystem method), 605  
uniform() (in module dask.array.random), 252  
unique() (dask.dataframe.Series method), 462  
unique() (in module dask.array), 179  
unpublish\_dataset() (distributed.Client method), 535  
unzip() (dask.bag.Bag method), 310  
upload\_file() (distributed.Client method), 535

## V

value\_counts() (dask.dataframe.Series method), 462  
values (dask.dataframe.DataFrame attribute), 399  
values (dask.dataframe.Series attribute), 462  
var() (dask.array.Array method), 283  
var() (dask.bag.Bag method), 310  
var() (dask.dataframe.DataFrame method), 399  
var() (dask.dataframe.groupby.DataFrameGroupBy method), 471  
var() (dask.dataframe.groupby.SeriesGroupBy method), 476  
var() (dask.dataframe.Series method), 462  
var() (in module dask.array), 180  
Variable (class in distributed), 538  
vdot() (in module dask.array), 181  
view() (dask.array.Array method), 284  
vindex (dask.array.Array attribute), 284  
visualize() (dask.dataframe.DataFrame method), 400  
visualize() (dask.dataframe.Series method), 463  
vnorm() (dask.array.Array method), 284  
vnorm() (in module dask.array), 182  
vonmises() (in module dask.array.random), 253  
vstack() (in module dask.array), 182

## W

wait() (in module distributed), 517  
wald() (in module dask.array.random), 254  
weibull() (in module dask.array.random), 255  
where() (dask.dataframe.DataFrame method), 401  
where() (dask.dataframe.Series method), 464  
where() (in module dask.array), 183  
who\_has() (distributed.Client method), 535

## Z

zeros() (in module dask.array), 184  
zeros\_like() (in module dask.array), 185  
zip() (in module dask.bag), 314  
zipf() (in module dask.array.random), 256