Dask is a flexible library for parallel computing in Python.

Dask is composed of two parts:

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 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](#) for more technical information on Dask’s distributed scheduler.
Familiar user interface

Dask DataFrame mimics Pandas - documentation

```python
import pandas as pd
import dask.dataframe as dd

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

Dask Array mimics NumPy - documentation

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

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

Dask Bag mimics iterators, Toolz, and PySpark - documentation

```python
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

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

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

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 the documentation about the distributed scheduler.

This ease of transition between single-machine to moderate cluster enables users to both start simple and grow when necessary.
Dask represents parallel computations with \textit{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 \texttt{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*
- *Community*
- *Why Dask?*

### 4.1 Install Dask

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

#### 4.1.1 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.
4.1.2 Pip

You can install everything required for most common uses of Dask (arrays, dataframes, ...) This installs both Dask and dependencies like NumPy, Pandas, and so on that are necessary for different workloads. This is often the right choice for Dask users:

```
pip install "dask[complete]"  # Install everything
```

You can also install only the Dask library. Modules like `dask.array`, `dask.dataframe`, `dask.delayed`, or `dask.distributed` won’t work until you also install NumPy, Pandas, Toolz, or Tornado, respectively. This is common for downstream library maintainers:

```
pip install dask               # Install only core parts of dask
```

We also maintain other dependency sets for different subsets of functionality:

```
pip install "dask[array]"      # Install requirements for dask array
pip install "dask[bag]"        # Install requirements for dask bag
pip install "dask[dataframe]"  # Install requirements for dask dataframe
pip install "dask[delayed]"    # Install requirements for dask delayed
pip install "dask[distributed]"# Install requirements for distributed dask
```

We have these options so that users of the lightweight core Dask scheduler aren’t required to download the more exotic dependencies of the collections (NumPy, Pandas, Tornado, etc.).

4.1.3 Install from Source

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

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

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

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

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

4.1.4 Anaconda

Dask is included by default in the Anaconda distribution.

4.1.5 Test

Test Dask with `py.test`:

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

Please be aware that installing Dask naively may not install all requirements by default. Please read the `pip` section above which discusses requirements. You may choose to install the `dask[complete]` version 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:
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. It 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`

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

```python
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 with the popular Kubernetes resource manager using either Helm or a native deployment.
  - **YARN / Hadoop**: Deploy Dask on YARN clusters, such as are found in traditional Hadoop installations.
  - **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.
Docker containers are available and may be useful in some of the solutions above.

Cloud for current recommendations on how to deploy Dask and Jupyter on common cloud providers like Amazon, Google, or Microsoft Azure.

## 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 it is sometimes useful with `dask.dataframe`

   Note that the `dask.distributed` scheduler is often a better choice when working with GIL-bound code. See [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

You can select between these options by specifying one of the following three values to the `scheduler=` keyword:

- "threads": Uses a ThreadPool in the local process
- "processes": Uses a ProcessPool to spread work between processes
- "single-threaded": Uses a for-loop in the current thread

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

- When calling `.compute()`
  
  ```python
  x.compute(scheduler='threads')
  ```

- With a context manager
  
  ```python
  with dask.config.set(scheduler='threads'):
      x.compute()
      y.compute()
  ```

- As a global setting
  
  ```python
  dask.config.set(scheduler='threads')
  ```

### Use the Distributed Scheduler

Dask’s newer 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.

```python
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

```python
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.

```python
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.

```python
from dask.distributed import Client, LocalCluster
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, like specifying explicit ports, and so on.
class distributed.deploy.local.LocalCluster(n_workers=None, threads_per_worker=None, processes=True, loop=None, start=None, ip=None, scheduler_port=0, silence_logs=30, dashboard_address=':8787', diagnostics_port=None, services=None, worker_services=None, service_kwargs=None, asynchronous=False, security=None, protocol=None, blocked_handlers=None, interface=None, worker_class=None, **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.WARN 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
- **dashboard_address**: str  Address on which to listen for the Bokeh diagnostics server like ‘localhost:8787’ or ‘0.0.0.0:8787’. Defaults to ‘:8787’. Set to None to disable the dashboard. Use port 0 for a random port.
- **diagnostics_port**: int  Deprecated. See dashboard_address.
- **asynchronous**: bool (False by default)  Set to True if using this cluster within async/await functions or within Tornado gen.coroutines. This should remain False for normal use.
- **worker_kwars**: dict  Extra worker arguments, will be passed to the Worker constructor.
- **blocked_handlers**: List[str]  A list of strings specifying a blacklist of handlers to disallow on the Scheduler, like ['feed', 'run_function']
- **service_kwars**: Dict[str, Dict]  Extra keywords to hand to the running services
- **security** [Security]
- **protocol**: str (optional)  Protocol to use like tcp://, tls://, inproc:// This defaults to sensible choice given other keyword arguments like processes and security
- **interface**: str (optional)  Network interface to use. Defaults to lo/localhost
- **worker_class**: Worker  Worker class used to instantiate workers from.

Examples

```python
>>> cluster = LocalCluster()  # Create a local cluster with as many workers as
˓
cores  # doctest: +SKIP
>>> cluster  # doctest: +SKIP
LocalCluster("127.0.0.1:8786", workers=8, ncores=8)
```
```python
>>> c = Client(cluster)  # connect to local cluster  # doctest: +SKIP
```

Add a new worker to the cluster

```python
>>> w = cluster.start_worker(ncores=2)  # doctest: +SKIP
```

Shut down the extra worker

```python
>>> cluster.stop_worker(w)  # doctest: +SKIP
```

Pass extra keyword arguments to Bokeh

```python
>>> LocalCluster(service_kwargs={'bokeh': {'prefix': '/foo'}})  # 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** *(**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

```python
>>> c = LocalCluster()  # doctest: +SKIP
```  

```python
>>> c.start_worker(ncores=2)  # doctest: +SKIP
```

**stop_worker** *(w)*
Stop a running worker

```
Examples

```python
>>> c = LocalCluster()  # doctest: +SKIP
```  

```python
>>> w = c.start_worker(ncores=2)  # doctest: +SKIP
```  

```python
>>> c.stop_worker(w)  # doctest: +SKIP
```
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. Hence, 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.

To accomplish this, 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).
Note:  For more information about relevant ports, please take a look 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

Warning: The command line documentation here may differ depending on your installed version. We recommend referring to the output of <command> --help.

dask-scheduler

dask-scheduler [OPTIONS] [PRELOAD_ARGV]...

Options

--host <host>
  URI, IP or hostname of this server

--port <port>
  Serving port

--interface <interface>
  Preferred network interface like ‘eth0’ or ‘ib0’

--tls-ca-file <tls_ca_file>
  CA cert(s) file for TLS (in PEM format)

--tls-cert <tls_cert>
  certificate file for TLS (in PEM format)

--tls-key <tls_key>
  private key file for TLS (in PEM format)

--bokeh-port <bokeh_port>
  Deprecated. See –dashboard-address

--dashboard-address <dashboard_address>
  Address on which to listen for diagnostics dashboard

--bokeh, --no-bokeh
  Launch Bokeh Web UI [default: True]

--show, --no-show
  Show web UI

--bokeh-whitelist <bokeh_whitelist>
  IP addresses to whitelist for bokeh.
--bokeh-prefix <bokeh_prefix>
    Prefix for the bokeh app

--use-xheaders <use_xheaders>
    User xheaders in bokeh app for ssl termination in header [default: False]

--pid-file <pid_file>
    File to write the process PID

--scheduler-file <scheduler_file>
    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 <local_directory>
    Directory to place scheduler files

--preload <preload>
    Module that should be loaded by the scheduler process like “foo.bar” or “/path/to/foo.py”.

Arguments

PRELOAD_ARGV
    Optional argument(s)

dask-worker

dask-worker [OPTIONS] [SCHEDULER] [PRELOAD_ARGV]...

Options

--tls-ca-file <tls_ca_file>
    CA cert(s) file for TLS (in PEM format)

--tls-cert <tls_cert>
    certificate file for TLS (in PEM format)

--tls-key <tls_key>
    private key file for TLS (in PEM format)

--worker-port <worker_port>
    Serving computation port, defaults to random

--nanny-port <nanny_port>
    Serving nanny port, defaults to random

--bokeh-port <bokeh_port>
    Deprecated. See --dashboard-address

--dashboard-address <dashboard_address>
    Address on which to listen for diagnostics dashboard

--bokeh, --no-bokeh
    Launch Bokeh Web UI [default: True]

--listen-address <listen_address>
    The address to which the worker binds. Example: tcp://0.0.0.0:9000
**--contact-address** `<contact_address>`
The address the worker advertises to the scheduler for communication with it and other workers. Example: tcp://127.0.0.1:9000

**--host** `<host>`
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** `<interface>`
Network interface like ‘eth0’ or ‘ib0’

**--nthreads** `<nthreads>`
Number of threads per process.

**--nprocs** `<nprocs>`
Number of worker processes to launch. Defaults to one.

**--name** `<name>`
A unique name for this worker like ‘worker-1’. If used with –nprocs then the process number will be appended like name-0, name-1, name-2, . . .

**--memory-limit** `<memory_limit>`
Bytes of memory per process 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** `<pid_file>`
File to write the process PID

**--local-directory** `<local_directory>`
Directory to place worker files

**--resources** `<resources>`
Resources for task constraints like “GPU=2 MEM=10e9”. Resources are applied separately to each worker process (only relevant when starting multiple worker processes with ‘–nprocs’).

**--scheduler-file** `<scheduler_file>`
Filename to JSON encoded scheduler information. Use with dask-scheduler –scheduler-file

**--death-timeout** `<death_timeout>`
Seconds to wait for a scheduler before closing

**--bokeh-prefix** `<bokeh_prefix>`
Prefix for the bokeh app

**--preload** `<preload>`
Module that should be loaded by each worker process like “foo.bar” or “/path/to/foo.py”

**Arguments**

SCHEDULER
Optional argument

PRELOAD_ARGV
Optional argument(s)
dask-ssh

Launch a distributed cluster over SSH. A ‘dask-scheduler’ process will run on the first host specified in [HOSTNAMES] or in the hostfile (unless --scheduler is specified explicitly). One or more ‘dask-worker’ processes will be run each host in [HOSTNAMES] or in the hostfile. Use command line flags to adjust how many dask-worker process are run on each host (--nprocs) and how many cpus are used by each dask-worker process (--nthreads).

dask-ssh [OPTIONS] [HOSTNAMES]...

Options

--scheduler <scheduler>
    Specify scheduler node. Defaults to first address.

--scheduler-port <scheduler_port>
    Specify scheduler port number. Defaults to port 8786.

--nthreads <nthreads>
    Number of threads per worker process. Defaults to number of cores divided by the number of processes per host.

--nprocs <nprocs>
    Number of worker processes per host. Defaults to one.

--hostfile <hostfile>
    Textfile with hostnames/IP addresses

--ssh-username <ssh_username>
    Username to use when establishing SSH connections.

--ssh-port <ssh_port>
    Port to use for SSH connections.

--ssh-private-key <ssh_private_key>
    Private key file to use for SSH connections.

--nohost
    Do not pass the hostname to the worker.

--log-directory <log_directory>
    Directory to use on all cluster nodes for the output of dask-scheduler and dask-worker commands.

--remote-python <remote_python>
    Path to Python on remote nodes.

--memory-limit <memory_limit>
    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

--worker-port <worker_port>
    Serving computation port, defaults to random

--nanny-port <nanny_port>
    Serving nanny port, defaults to random

--remote-dask-worker <remote_dask_worker>
    Worker to run. Defaults to distributed.cli.dask_worker
Arguments

HOSTNAMES
  Optional argument(s)

dask-submit

dask-submit [OPTIONS] REMOTE_CLIENT_ADDRESS FILEPATH

Arguments

REMOTE_CLIENT_ADDRESS
  Required argument

FILEPATH
  Required argument

dask-remote

dask-remote [OPTIONS]

Options

--host <host>
  IP or hostname of this server

--port <port>
  Remote Client Port

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:
CLI Options

Launch a distributed cluster over SSH. A `dask-scheduler` process will run on the first host specified in `[HOSTNAMES]` or in the hostfile (unless `--scheduler` is specified explicitly). One or more `dask-worker` processes will be run on each host in `[HOSTNAMES]` or in the hostfile. Use command line flags to adjust how many `dask-worker` processes are run on each host (`--nprocs`) and how many cpus are used by each `dask-worker` process (`--nthreads`).

Options:

**Note:** This table may grow out of date, you should check `dask-ssh --help` to get an up-to-date listing of all options.

<table>
<thead>
<tr>
<th>Option</th>
<th>TYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--scheduler</code></td>
<td>TEXT</td>
<td>Specify scheduler node. Defaults to first address</td>
</tr>
<tr>
<td><code>--scheduler-port</code></td>
<td>INTEGER</td>
<td>Specify scheduler port number. Defaults to port 8786</td>
</tr>
<tr>
<td><code>--nthreads</code></td>
<td>INTEGER</td>
<td>Number of threads per worker process. Defaults to number of cores divided by the number of processes per host</td>
</tr>
<tr>
<td><code>--nprocs</code></td>
<td>INTEGER</td>
<td>Number of worker processes per host. Defaults to one</td>
</tr>
<tr>
<td><code>--host-file</code></td>
<td>PATH</td>
<td>Textfile with hostnames/IP addresses</td>
</tr>
<tr>
<td><code>--ssh-username</code></td>
<td>TEXT</td>
<td>Username to use when establishing SSH connections</td>
</tr>
<tr>
<td><code>--ssh-port</code></td>
<td>INTEGER</td>
<td>Port to use for SSH connections</td>
</tr>
<tr>
<td><code>--ssh-private-key</code></td>
<td>TEXT</td>
<td>Private key file to use for SSH connections</td>
</tr>
<tr>
<td><code>--log-directory</code></td>
<td>PATH</td>
<td>Directory to use on all cluster nodes for the output of dask-scheduler and dask-worker commands</td>
</tr>
<tr>
<td><code>--remote-python</code></td>
<td>TEXT</td>
<td>Path to Python on remote nodes</td>
</tr>
<tr>
<td><code>--memory-limit</code></td>
<td>TEXT</td>
<td>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</td>
</tr>
<tr>
<td><code>--worker-port</code></td>
<td>INTEGER</td>
<td>Serving computation port, defaults to random</td>
</tr>
<tr>
<td><code>--nanny-port</code></td>
<td>INTEGER</td>
<td>Serving nanny port, defaults to random</td>
</tr>
<tr>
<td><code>--nohost</code></td>
<td></td>
<td>Do not pass the hostname to the worker</td>
</tr>
</tbody>
</table>
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

Where to start

Most of this page documents various ways and best practices to use Dask on an HPC cluster. This is technical and aimed both at users with some experience deploying Dask and also system administrators.

The preferred and simplest way to run Dask on HPC systems today both for new, experienced users or administrator is to use dask-jobqueue.

However, dask-jobqueue is slightly oriented toward interactive analysis usage, and it might be better to use tools like dask-mpi in some routine batch production workloads.

Dask-jobqueue and Dask-drmaa

The following projects provide easy high-level access to Dask using resource managers that are commonly deployed on HPC systems:

1. dask-jobqueue for use with PBS, SLURM, LSF, SGE and other resource managers
2. dask-drmaa for use with any DRMAA compliant resource manager

They provide interfaces that look like the following:

```python
from dask_jobqueue import PBSCluster
cluster = PBSCluster(cores=36,
                     memory="100GB",
                     project='P48500028',
                     queue='premium',
                     interface='ib0',
                     walltime='02:00:00')
cluster.scale(100)  # Start 100 workers in 100 jobs that match the description above
from dask.distributed import Client
client = Client(cluster)  # Connect to that cluster
```

Dask-jobqueue provides a lot of possibilities like adaptive dynamic scaling of workers, we recommend reading the dask-jobqueue documentation first to get a basic system running and then returning to this documentation for fine-tuning if necessary.
Using MPI

Note: This section may not be necessary if you use a tool like dask-jobqueue.

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 or open-mpi 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.

In some setups, MPI processes are not allowed to fork other processes. In this case, we recommend using `--no-nanny` option in order to prevent dask from using an additional nanny process to manage workers.

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

Using a Shared Network File System and a Job Scheduler

Note: This section is not necessary if you use a tool like dask-jobqueue.

Some clusters benefit from a shared File System (NFS, GPFS, Lustre or alike), 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 the connection information to a 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.
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 with the `dask-worker`, `dask-scheduler`, or `dask-mpi` commands or the `interface=` keyword with the `dask-jobqueue Cluster` objects:

```
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
```


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 in your `~/.config/dask/distributed.yaml` file:

```
distributed:
  worker:
    memory:
      target: false  # don't spill to disk
      spill: false  # don't spill to disk
      pause: 0.80   # pause execution at 80% memory use
      terminate: 0.95  # restart the worker at 95% use
```

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
```

4.2. Setup
Launch Many Small Jobs

Note: This section is not necessary if you use a tool like dask-jobqueue.

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. If we split 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:

```python
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)
```

4.2.6 Kubernetes

Kubernetes and Helm

It is easy to launch a Dask cluster and a Jupyter notebook server on cloud resources using Kubernetes and Helm. This is particularly useful when you want to deploy a fresh Python environment on Cloud services like Amazon Web Services, Google Compute Engine, or Microsoft Azure.

If you already have Python environments running in a pre-existing Kubernetes cluster, then you may prefer the Kubernetes native documentation, which is a bit lighter weight.

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 on one of the common cloud providers like Google, Amazon, or Microsoft. 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). Also, 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 chart in the default stable channel at https://kubernetes-charts.storage.googleapis.com. This should be added to your helm installation by default. You can update the known channels to make sure you have up-to-date charts as follows:

```
helm repo update
```

Now, you can launch Dask on your Kubernetes cluster using the Dask Helm chart:

```
helm install stable/dask
```

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

## Verify Deployment

This might take a minute to deploy. You can check its status with `kubectl`:

```
$ kubectl get pods
NAME                      READY STATUS      RESTARTS AGE
bald-eel-jupyter-924045334-twxtd 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-b8nqq 1/1  Running 0 1m
bald-eel-worker-3032746726-d0chx 0/1  ContainerCreating 0 1m

$ kubectl get services
NAME     TYPE             CLUSTER-IP     EXTERNAL-IP      PORT(S)
---      -------           --------          ---------------      ------
bald-eel-jupyter  LoadBalancer  10.11.247.201  35.226.183.149  80:30173/TCP
bald-eel-scheduler  LoadBalancer  10.11.245.241  35.202.201.129  8786:31166/TCP,
                  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. Note that you will need to use this name to refer to your deployment in the future. Additionally, you can list all active helm deployments with:

```
helm list
```

(continues on next page)
Connect to Dask and Jupyter

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

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

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

We can navigate to these services from any web browser. Here, one is the Dask diagnostic dashboard, and 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']
'bald-eel-scheduler:8786'
```

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

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

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 yaml file that implements a subset of the values in the `dask helm chart values.yaml` 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
  resources:
    limits:
      cpu: 2
      memory: 7.5G
      requests:
```

(continues on next page)
This config file overrides the configuration for the 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 stable/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 the worker status and logs using the Dask dashboard (in particular, you can see the worker links from the info/ page). However, if your workers aren’t starting, you can check 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-gsgj7       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.
```

(continues on next page)
Delete a Helm deployment

You can always delete a helm deployment using its name:

```
helm delete bald-eel --purge
```

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.

```
jupyter:
  enabled: false
```

Kubernetes Native

See external documentation on Dask-Kubernetes for more information.

Kubernetes is a popular system for deploying distributed applications on clusters, particularly in the cloud. You can use Kubernetes to launch Dask workers in the following two ways:

1. Helm: You can launch a Dask scheduler, several workers, and an optional Jupyter Notebook server on a Kubernetes easily using Helm

```
helm repo update  # get latest helm charts
helm install stable/dask  # deploy standard dask chart
```

This is a good choice if you want to do the following:

1. Run a managed Dask cluster for a long period of time
2. Also deploy a Jupyter server from which to run code
3. Share the same Dask cluster between many automated services
4. Try out Dask for the first time on a cloud-based system like Amazon, Google, or Microsoft Azure (see also our Cloud documentation)

Note: For more information, see Dask and Helm documentation.
2. **Native**: You can quickly deploy Dask workers on Kubernetes from within a Python script or interactive session using Dask-Kubernetes

```python
from dask_kubernetes import KubeCluster
cluster = KubeCluster.from_yaml('worker-template.yaml')
cluster.scale(20)  # add 20 workers
cluster.adapt()  # or create and destroy workers dynamically based on workload

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

This is a good choice if you want to do the following:

1. Dynamically create a personal and ephemeral deployment for interactive use
2. Allow many individuals the ability to launch their own custom dask deployments, rather than depend on a centralized system
3. Quickly adapt Dask cluster size to the current workload

**Note**: For more information, see Dask-Kubernetes documentation.

You may also want to see the documentation on using *Dask with Docker containers* to help you manage your software environments on Kubernetes.

### 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**

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

```python
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 this case, you would replace the `loop.start()` call with the following:

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

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

```python
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. Also, it provides some additional monitoring, and is useful when coordinating many workers that should live in different processes in order to avoid the GIL.

### 4.2.8 Cloud Deployments

To get started running Dask on common Cloud providers like Amazon, Google, or Microsoft, we currently recommend deploying *Dask with Kubernetes and Helm*. All three major cloud vendors now provide managed Kubernetes services. This allows us to reliably provide the same experience across all clouds, and ensures that solutions for any one provider remain up-to-date.

Alternatively, if you are deploying on a cloud-hosted Hadoop cluster like Amazon EMR or Google Cloud DataProc, you will want to use Dask-Yarn. Documentation on deploying on Amazon EMR specifically can be found here, the process is similar for Google Cloud DataProc.

**Data Access**

You may want to install additional libraries in your Jupyter and worker images to access the object stores of each cloud:

- `s3fs` for Amazon’s S3
- `gcsfs` for Google’s GCS
- `adlfs` for Microsoft’s ADL

**Historical Libraries**

Dask previously maintained libraries for deploying Dask on Amazon’s EC2. Due to sporadic interest, and churn both within the Dask library and EC2 itself, these were not well maintained. They have since been deprecated in favor of the *Kubernetes and Helm* solution.

### 4.2.9 Adaptive Deployments

**Motivation**

Most Dask deployments are static with a single scheduler and a fixed number of workers. This results in predictable behavior, but is wasteful of resources in two situations:
1. The user may not be using the cluster, or perhaps they are busy interpreting a recent result or plot, and so the workers sit idly, taking up valuable shared resources from other potential users.

2. The user may be very active, and is limited by their original allocation.

Particularly efficient users may learn to manually add and remove workers during their session, but this is rare. Instead, we would like the size of a Dask cluster to match the computational needs at any given time. This is the goal of the adaptive deployments discussed in this document. These are particularly helpful for interactive workloads, which are characterized by long periods of inactivity interrupted with short bursts of heavy activity. Adaptive deployments can result in both faster analyses that give users much more power, but with much less pressure on computational resources.

**Adaptive**

To make setting up adaptive deployments easy, some Dask deployment solutions offer an `.adapt()` method. Here is an example with `dask_kubernetes.KubeCluster`.

```python
from dask_kubernetes import KubeCluster
cluster = KubeCluster()
cluster.adapt(minimum=0, maximum=100)  # scale between 0 and 100 workers
```

For more keyword options, see the Adaptive class below:

```python
Adaptive(scheduler[, cluster, interval, ...]) Adaptively allocate workers based on scheduler load.
```

**Dependence on a Resource Manager**

The Dask scheduler does not know how to launch workers on its own. Instead, it relies on an external resource scheduler like Kubernetes above, or Yarn, SGE, SLURM, Mesos, or some other in-house system (see setup documentation for options). In order to use adaptive deployments, you must provide some mechanism for the scheduler to launch new workers. Typically, this is done by using one of the solutions listed in the setup documentation, or by subclassing from the Cluster superclass and implementing that API.

**Scaling Heuristics**

The Dask scheduler tracks a variety of information that is useful to correctly allocate the number of workers:

1. The historical runtime of every function and task that it has seen, and all of the functions that it is currently able to run for users

2. The amount of memory used and available on each worker

3. Which workers are idle or saturated for various reasons, like the presence of specialized hardware

From these, it is able to determine a target number of workers by dividing the cumulative expected runtime of all pending tasks by the `target_duration` parameter (defaults to five seconds). This number of workers serves as a baseline request for the resource manager. This number can be altered for a variety of reasons:

1. If the cluster needs more memory, then it will choose either the target number of workers or twice the current number of workers (whichever is larger)
2. If the target is outside of the range of the minimum and maximum values, then it is clipped to fit within that range.

Additionally, when scaling down, Dask preferentially chooses those workers that are idle and have the least data in memory. It moves that data to other machines before retiring the worker. To avoid rapid cycling of the cluster up and down in size, we only retire a worker after a few cycles have gone by where it has consistently been a good idea to retire it (controlled by the `wait_count` and `interval` parameters).

**API**

```python
class distributed.deploy.Adaptive(scheduler, cluster=None, interval='1s',
                                startup_cost='1s', scale_factor=2, minimum=0, maximum=None,
                                wait_count=3, target_duration='5s',
                                worker_key=<function Adaptive.<lambda>>, **kwargs)
```

Adaptively allocate workers based on scheduler load. A superclass.

Contains logic to dynamically resize a Dask cluster based on current use. This class needs to be paired with a system that can create and destroy Dask workers using a cluster resource manager. Typically it is built into already existing solutions, rather than used directly by users. It is most commonly used from the `.adapt(., .)` method of various Dask cluster classes.

**Parameters**

- `scheduler`: distributed.Scheduler
- `cluster`: object  Must have `scale_up` and `scale_down` methods/coroutines
- `startup_cost` [timedelta or str, default “1s”] Estimate of the number of seconds for nnFactor representing how costly it is to start an additional worker. Affects quickly to adapt to high tasks per worker loads
- `interval` [timedelta or str, default “1000 ms”] Milliseconds between checks
- `wait_count`: int, default 3  Number of consecutive times that a worker should be suggested for removal before we remove it.
- `scale_factor` [int, default 2] Factor to scale by when it’s determined additional workers are needed
- `target_duration`: timedelta or str, default “5s” Amount of time we want a computation to take. This affects how aggressively we scale up.
- `worker_key`: Callable[WorkerState] Function to group workers together when scaling down
  See Scheduler.workers_to_close for more information
- `minimum`: int  Minimum number of workers to keep around
- `maximum`: int  Maximum number of workers to keep around
- `**kwargs`: Extra parameters to pass to Scheduler.workers_to_close

**Notes**

Subclasses can override `Adaptive.should_scale_up()` and `Adaptive.workers_to_close()` to control when the cluster should be resized. The default implementation checks if there are too many tasks per worker or too little memory available (see `Adaptive.needs_cpu()` and `Adaptive.needs_memory()`).

`Adaptive.get_scale_up_kwargs()` method controls the arguments passed to the cluster’s `scale_up` method.
Examples

This is commonly used from existing Dask classes, like KubeCluster

```python
>>> from dask_kubernetes import KubeCluster
>>> cluster = KubeCluster()
>>> cluster.adapt(minimum=10, maximum=100)
```

Alternatively you can use it from your own Cluster class by subclassing from Dask’s Cluster superclass

```python
>>> from distributed.deploy import Cluster
>>> class MyCluster(Cluster):
...     def scale_up(self, n):
...         ...         """ Bring worker count up to n """
...     def scale_down(self, workers):
...         ...         """ Remove worker addresses from cluster """

>>> cluster = MyCluster()
>>> cluster.adapt(minimum=10, maximum=100)
```

class distributed.deploy.Cluster
Superclass for cluster objects

This expects a local Scheduler defined on the object. It provides common methods and an IPython widget display.

Clusters inheriting from this class should provide the following:

1. A local Scheduler object at .scheduler
2. scale_up and scale_down methods as defined below:

   ```python
def scale_up(self, n: int):
        """ Bring total worker count up to n """
def scale_down(self, workers: List[str]):
        """ Close the workers with the given addresses """
```

This will provide a general scale method as well as an IPython widget for display.

See also:

LocalCluster a simple implementation with local workers

Examples

```python
>>> from distributed.deploy import Cluster
>>> class MyCluster(Cluster):
...     def scale_up(self, n):
...         ...         """ Bring the total worker count up to n """
...     pass
...     def scale_down(self, workers):
...         ...         """ Close the workers with the given addresses """
...     pass

>>> cluster = MyCluster()
>>> cluster.scale(5)              # scale manually
>>> cluster.adapt(minimum=1, maximum=100)  # scale automatically
```
4.2.10 Docker Images

Example docker images are maintained at https://github.com/dask/dask-docker and https://hub.docker.com/r/daskdev/.

Each image installs the full Dask conda package (including the distributed scheduler), Numpy, and Pandas on top of a Miniconda installation on top of a Debian image.

These images are large, around 1GB.

- **daskdev/dask**: This a normal debian + miniconda image with the full Dask conda package (including the distributed scheduler), Numpy, and Pandas. This image is about 1GB in size.
- **daskdev/dask-notebook**: This is based on the Jupyter base-notebook image and so it is suitable for use both normally as a Jupyter server, and also as part of a JupyterHub deployment. It also includes a matching Dask software environment described above. This image is about 2GB in size.

**Example**

Here is a simple example on the local host network

```bash
docker run -it --network host daskdev/dask dask-scheduler  # start scheduler
docker run -it --network host daskdev/dask dask-worker localhost:8786  # start worker
docker run -it --network host daskdev/dask dask-worker localhost:8786  # start worker
docker run -it --network host daskdev/dask dask-worker localhost:8786  # start worker
docker run -it --network host daskdev/dask-notebook  # start Jupyter server
```

**Extensibility**

Users can mildly customize the software environment by populating the environment variables `EXTRA_APT_PACKAGES`, `EXTRA_CONDA_PACKAGES`, and `EXTRA_PIP_PACKAGES`. If these environment variables are set, they will trigger calls to the following respectively:

```
apt-get install $EXTRA_APT_PACKAGES
conda install $EXTRA_CONDA_PACKAGES
pip install $EXTRA_PIP_PACKAGES
```

Note that using these can significantly delay the container from starting, especially when using `apt`, or `conda` (pip is relatively fast).

Remember that it is important for software versions to match between Dask workers and Dask clients. As a result, it is often useful to include the same extra packages in both Jupyter and Worker images.

**Source**

Docker files are maintained at https://github.com/dask/dask-docker. This repository also includes a docker-compose configuration.

4.2.11 Custom Initialization

Often we want to run custom code when we start up or tear down a scheduler or worker. We might do this manually with functions like `Client.run` or `Client.run_on_scheduler`, but this is error prone and difficult to automate.
To resolve this, Dask includes a few mechanisms to run arbitrary code around the lifecycle of a Scheduler or Worker.

**Preload Scripts**

Both `dask-scheduler` and `dask-worker` support a `--preload` option that allows custom initialization of each scheduler/worker respectively. A module or Python file passed as a `--preload` value is guaranteed to be imported before establishing any connection. A `dask_setup(service)` function is called if found, with a `Scheduler` or `Worker` instance as the argument. As the service stops, `dask_teardown(service)` is called if present.

To support additional configuration, a single `--preload` module may register additional command-line arguments by exposing `dask_setup` as a `Click` command. This command will be used to parse additional arguments provided to `dask-worker` or `dask-scheduler` and will be called before service initialization.

As an example, consider the following file that creates a scheduler plugin and registers it with the scheduler

```python
# scheduler-setup.py
import clickrom distributed.diagnostics.plugin import SchedulerPlugin
class MyPlugin(SchedulerPlugin):
    def __init__(self, print_count):
        self.print_count = print_count
        SchedulerPlugin.__init__(self)
        def add_worker(self, scheduler=None, worker=None, **kwargs):
            print("Added a new worker at:", worker)
            if self.print_count and scheduler is not None:
                print("Total workers:", len(scheduler.workers))

@click.command()
@click.option("--print-count/--no-print-count", default=False)
def dask_setup(scheduler, print_count):
    plugin = MyPlugin(print_count)
    scheduler.add_plugin(plugin)
```

We can then run this preload script by referring to its filename (or module name if it is on the path) when we start the scheduler:

```
dask-scheduler --preload scheduler-setup.py --print-count
```

**Worker Lifecycle Plugins**

You can also create a class with setup and teardown methods, and register that class with the scheduler to give to every worker.

```
Client.register_worker_plugin
```

### 4.3 Use Cases

Dask is a versatile tool that supports a variety of workloads. This page contains brief and illustrative examples of 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 where Dask is used 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 more useful to individuals (more people have personal laptops than have access to clusters) and probably accounts for 80+% of the use of Dask today. On the other hand, the distributed machine scheduler is more 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, compute seasonal averaging, etc., all with straight NumPy syntax. These computations take a few minutes to execute (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.

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

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

(continues on next page)
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 other 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, it can be a slow process. So, they can use dask.dataframe to logically wrap all of these different files into one logical DataFrame that is built on demand to save space. Since most of their Pandas workflow is the same (Dask’s DataFrame is a subset of Pandas), they can switch from Pandas to Dask and back easily without significantly changing their code.

```python
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 analysts as above use dask.dataframe with the dask.distributed scheduler to analyze terabytes of data on their institution’s Hadoop cluster straight from Python. This uses either the hdfs3 or pyarrow Python libraries 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 which the analyst is already very comfortable with.

```python
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 analysts also have 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 their data (such as the date or other metadata). To work around this, they use dask.delayed to teach dask.dataframe how to load the data and then pass it 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**

Some programmers have 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. To accomplish this, they wrap their function in `dask.delayed` and then let the appropriate dask scheduler parallelize and load balance the work.

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

**Normal Sequential Processing:**

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

**Build Dask Computation:**

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

**Multiple Threads:**

```python
import dask.threaded
results = compute(*values, scheduler='threads')
```

**Multiple Processes:**

```python
import dask.multiprocessing
results = compute(*values, scheduler='processes')
```

**Distributed Cluster:**

```python
from dask.distributed import Client
client = Client("cluster-address:8786")
results = compute(*values, scheduler='distributed')
```
Complex dependencies

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

```python
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. Now, the analyst has his code as a collection of Python functions, and is trying to figure out how to parallelize such a codebase. To solve this, he uses `dask.delayed` to wrap his function calls and capture the implicit parallelism.

```python
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)
```

The analyst then depends on the dask schedulers to run this complex web of computations in parallel.

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

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

Algorithm developer

A couple of graduate students in machine learning are 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` 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 on the cloud` to easily provision clusters of varying sizes.

Their algorithm is written in the same way in all cases. This drastically reduces the cognitive load, and lets 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 her machine learning pipeline to run on a cluster to accelerate parameter searches. She already uses the `sklearn njobs=` parameter to accelerate computations on her local computer with Joblib. Now, she wraps her `sklearn` code with a context manager to parallelize the exact same code across a cluster (also available with IPyParallel)

```python
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 to be 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. He contacts these users and teaches 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. She combines `dask.distributed` with normal Python Queues to produce a rudimentary but effective stream processing system. Because `dask.distributed` is elastic, she can scale up or scale down her cluster resources in response to demand.

**4.4 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.

**4.4.1 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` for general conversation and `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. **Monthly developer meeting** happens the first Thursday of the month at 4pm UTC (11am in New York, 8am in Los Angeles, 12am in Beijing) at `https://appear.in/dask-dev`

**4.4.2 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**: We strongly prefer the use of Stack Overflow or GitHub issues over Gitter chat. GitHub and Stack Overflow are more easily searchable by future users, and therefore is more efficient for everyone’s time. Gitter chat is strictly reserved for developer and community discussion.

   If you have a general question about how something should work or want best practices then use Stack Overflow. If you think you have found a bug then use GitHub.

2. **Ask only in one place**: Please restrict yourself to posting your question in only one place (likely Stack Overflow or GitHub) and don’t post in both.

3. **Create a minimal example**: It is ideal to create minimal, complete, verifiable examples. This significantly reduces the time that answerers spend understanding your situation, resulting in higher quality answers more quickly.

   See also this blogpost about crafting minimal bug reports. These have a much higher likelihood of being answered.

**4.4.3 Paid support**

In addition to the previous options, paid support is available from

- Anaconda: `https://www.anaconda.com/support`
4.5 Why Dask?

This document gives high-level motivation on why people choose to adopt Dask.

4.5.1 Python’s role in Data Science

Python has grown to become the dominant language both in data analytics and general programming:

![Growth of major programming languages](image)

This is fueled both by computational libraries like Numpy, Pandas, and Scikit-Learn and by a wealth of libraries for visualization, interactive notebooks, collaboration, and so forth.
However, these packages were not designed to scale beyond a single machine. Dask was developed to scale these packages and the surrounding ecosystem. It works with the existing Python ecosystem to scale it to multi-core machines and distributed clusters.

*Image credit to Stack Overflow blogposts #1 and #2.*

### 4.5.2 Familiar API

Analysts often use tools like Pandas, Scikit-Learn, Numpy, and the rest of the Python ecosystem to analyze data on their personal computer. They like these tools because they are efficient, intuitive, and widely trusted. However, when they choose to apply their analyses to larger datasets, they find that these tools were not designed to scale beyond a single machine. Therefore, the analyst is forced to rewrite their computation using a more scalable tool, often in another language altogether. This rewrite process slows down discovery and causes frustration.

Dask provides ways to scale Pandas, Scikit-Learn, and Numpy workflows with minimal rewriting. It integrates well with these tools so that it copies most of their API and uses their data structures internally. Moreover, Dask is co-developed with these libraries to ensure that they evolve consistently, minimizing friction caused from transitioning from workloads on a local laptop, to a multi-core workstation, and to a distributed cluster. Analysts familiar with Pandas/Scikit-Learn/Numpy will be immediately familiar with their Dask equivalents, and have much of their intuition carry over to a scalable context.
4.5.3 Scales out to clusters

As datasets and computations scale faster than CPUs and RAM, we need to find ways to scale our computations across multiple machines. This introduces many new concerns:

- How to have computers talk to each other over the network?
- How and when to move data between machines?
- How to recover from machine failures?
- How to deploy on an in-house cluster?
- How to deploy on the cloud?
- How to deploy on an HPC super-computer?
- How to provide an API to this system that users find intuitive?
- ...

While it is possible to build these systems in-house (and indeed, many exist), many organizations are increasingly depending on solutions developed within the open source community. These tend to be more robust, secure, and fully featured without being tended by in-house staff.

Dask solves these problems. It is routinely run on thousand-machine clusters to process hundreds of terabytes of data efficiently. It has utilities and documentation on how to deploy in-house, on the cloud, or on HPC super-computers. It supports encryption and authentication using TLS/SSL certificates. It is resilient and can handle the failure of worker nodes gracefully and is elastic, and so can take advantage of new nodes added on-the-fly. Dask includes several user APIs that are used and smoothed over by thousands of researchers across the globe working in different domains.

4.5.4 Scales down to single computers

But a massive cluster is not always the right choice

Today’s laptops and workstations are surprisingly powerful and, if used correctly, can often handle datasets and computations for which we previously depended on clusters. A modern laptop has a multi-core CPU, 32GB of RAM, and flash-based hard drives that can stream through data several times faster than HDDs or SSDs of even a year or two ago.

As a result, analysts can often manipulate 100GB+ datasets on their laptop or 1TB+ datasets on a workstation without bothering with the cluster at all. They sometimes prefer this for the following reasons:

1. They can use their local software environment, rather than being constrained by what is available on the cluster
2. They can more easily work while in transit, at a coffee shop, or at home away from the VPN
3. Debugging errors and analyzing performance are generally much easier on a single machine without having to pore through logs
4. Generally their iteration cycles are faster
5. Their computations may be more efficient because all of the data is local and doesn’t need to flow through the network or between separate processes

Dask can enable efficient parallel computations on single machines by leveraging their multi-core CPUs and streaming data efficiently from disk. It can run on a distributed cluster, but it doesn’t have to. Dask allows you to swap out the cluster for single-machine schedulers which are surprisingly lightweight, require no setup, and can run entirely within the same process as the user’s session.

To avoid excess memory use, Dask is good at finding ways to evaluate computations in a low-memory footprint when possible by pulling in chunks of data from disk, doing the necessary processing, and throwing away intermediate
values as quickly as possible. This lets analysts perform computations on moderately large datasets (100GB+) even on relatively low-power laptops. This requires no configuration and no setup, meaning that adding Dask to a single-machine computation adds very little cognitive overhead.

### 4.5.5 Integrates with the Python ecosystem

Python includes computational libraries like Numpy, Pandas, and Scikit-Learn, along with thousands of others in data access, plotting, statistics, image and signal processing, and more. These libraries work together seamlessly to produce a cohesive ecosystem of packages that co-evolve to meet the needs of analysts in many domains.

This ecosystem is tied together by common standards and protocols to which everyone adheres, which allows these packages to benefit each other in surprising and delightful ways.

Dask evolved from within this ecosystem. It abides by these standards and protocols and actively engages in community efforts to push forward new ones. This enables the rest of the ecosystem to benefit from parallel and distributed computing with minimal coordination. Dask does not seek to disrupt or displace the existing ecosystem, but rather to complement and benefit it from within.

As a result, Dask development is pushed forward by developer communities from Pandas, Numpy, Scikit-Learn, Scikit-Image, Jupyter, and others. This engagement from the broader community growth helps users to trust the project and helps to ensure that the Python ecosystem will continue to evolve in a smooth and sustainable manner.

### 4.5.6 Supports complex applications

Some parallel computations are simple and just apply the same routine onto many inputs without any kind of coordination. These are simple to parallelize with any system.

Somewhat more complex computations can be expressed with the map-shuffle-reduce pattern popularized by Hadoop and Spark. This is often sufficient to do most data cleaning tasks, database-style queries, and some lightweight machine learning algorithms.

However, more complex parallel computations exist which do not fit into these paradigms, and so are difficult to perform with traditional big-data technologies. These include more advanced algorithms for statistics or machine learning, time series or local operations, or bespoke parallelism often found within the systems of large enterprises.

Many companies and institutions today have problems which are clearly parallelizable, but not clearly transformable into a big DataFrame computation. Today these companies tend to solve their problems either by writing custom code with low-level systems like MPI, ZeroMQ, or sockets and complex queuing systems, or by shoving their problem into a standard big-data technology like MapReduce or Spark, and hoping for the best.

Dask helps to resolve these situations by exposing low-level APIs to its internal task scheduler which is capable of executing very advanced computations. This gives engineers within the institution the ability to build their own parallel computing system using the same engine that powers Dask’s arrays, DataFrames, and machine learning algorithms, but now with the institution’s own custom logic. This allows engineers to keep complex business logic in-house while still relying on Dask to handle network communication, load balancing, resilience, diagnostics, etc..

### 4.5.7 Responsive feedback

Because everything happens remotely, interactive parallel computing can be frustrating for users. They don’t have a good sense of how computations are progressing, what might be going wrong, or what parts of their code should they focus on for performance. The added distance between a user and their computation can drastically affect how quickly they are able to identify and resolve bugs and performance problems, which can drastically increase their time to solution.
Dask keeps users informed and content with a suite of helpful diagnostic and investigative tools including the following:

1. A **real-time and responsive dashboard** that shows current progress, communication costs, memory use, and more, updated every 100ms
2. A statistical profiler installed on every worker that polls each thread every 10ms to determine which lines in your code are taking up the most time across your entire computation
3. An embedded IPython kernel in every worker and the scheduler, allowing users to directly investigate the state of their computation with a pop-up terminal
4. The ability to reraise errors locally, so that they can use the traditional debugging tools to which they are accustomed, even when the error happens remotely

**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 User Interfaces

Dask supports several user interfaces:

- **High-Level**
  - **Arrays**: Parallel NumPy
  - **Bags**: Parallel lists
  - **DataFrames**: Parallel Pandas
  - **Machine Learning**: Parallel Scikit-Learn
  - Others from external projects, like XArray

- **Low-Level**
  - **Delayed**: Parallel function evaluation
  - **Futures**: Real-time parallel function evaluation

Each of these user interfaces employs the same underlying parallel computing machinery, and so has the same scaling, diagnostics, resilience, and so on, but each provides a different set of parallel algorithms and programming style.

This document helps you to decide which user interface best suits your needs, and gives some general information that applies to all interfaces. The pages linked above give more information about each interface in greater depth.
4.6.1 High-Level Collections

Many people who start using Dask are explicitly looking for a scalable version of NumPy, Pandas, or Scikit-Learn. For these situations, the starting point within Dask is usually fairly clear. If you want scalable NumPy arrays, then start with Dask array; if you want scalable Pandas DataFrames, then start with Dask DataFrame, and so on.

These high-level interfaces copy the standard interface with slight variations. These interfaces automatically parallelize over larger datasets for you for a large subset of the API from the original project.

```python
# Arrays
import dask.array as da
x = da.random.uniform(low=0, high=10, size=(10000, 10000), # normal numpy code
                      chunks=(1000, 1000))  # break into chunks of size 1000x1000
y = x + x.T - x.mean(axis=0)  # Use normal syntax for high level algorithms

# DataFrames
import dask.dataframe as dd
df = dd.read_csv('2018-*-*.csv', parse_dates='timestamp', # normal pandas code
                 blocksize=64000000)  # break text into 64MB chunks
s = df.groupby('name').balance.mean()  # Use normal syntax for high level algorithms

# Bags / lists
import dask.bag as db
b = db.read_text('*.json').map(json.loads)
total = (b.filter(lambda d: d['name'] == 'Alice')
          .map(lambda d: d['balance'])
          .sum())
```

It is important to remember that, while APIs may be similar, some differences do exist. Additionally, the performance of some algorithms may differ from their in-memory counterparts due to the advantages and disadvantages of parallel programming. Some thought and attention is still required when using Dask.

4.6.2 Low-Level Interfaces

Often when parallelizing existing code bases or building custom algorithms, you run into code that is parallelizable, but isn’t just a big DataFrame or array. Consider the for-loopy code below:

```python
results = []
for a in A:
    for b in B:
        if a < b:
            c = f(a, b)
        else:
            c = g(a, b)
        results.append(c)
```

There is potential parallelism in this code (the many calls to f and g can be done in parallel), but it’s not clear how to rewrite it into a big array or DataFrame so that it can use a higher-level API. Even if you could rewrite it into one of these paradigms, it’s not clear that this would be a good idea. Much of the meaning would likely be lost in translation, and this process would become much more difficult for more complex systems.

Instead, Dask’s lower-level APIs let you write parallel code one function call at a time within the context of your existing for loops. A common solution here is to use Dask delayed to wrap individual function calls into a lazily constructed task graph:
import dask

lazy_results = []
for a in A:
    for b in B:
        if a < b:
            c = dask.delayed(f)(a, b)  # add lazy task
        else:
            c = dask.delayed(g)(a, b)  # add lazy task
    lazy_results.append(c)

results = dask.compute(*lazy_results)  # compute all in parallel

4.6.3 Combining High- and Low-Level Interfaces

It is common to combine high- and low-level interfaces. For example, you might use Dask array/bag/dataframe to load in data and do initial pre-processing, then switch to Dask delayed for a custom algorithm that is specific to your domain, then switch back to Dask array/dataframe to clean up and store results. Understanding both sets of user interfaces, and how to switch between them, can be a productive combination.

# Convert to a list of delayed Pandas dataframes
delayed_values = df.to_delayed()

# Manipulate delayed values arbitrarily as you like

# Convert many delayed Pandas DataFrames back to a single Dask DataFrame
df = dd.from_delayed(delayed_values)

4.6.4 Laziness and Computing

Most Dask user interfaces are lazy, meaning that they do not evaluate until you explicitly ask for a result using the compute method:

# This array syntax doesn't cause computation
y = x + x.T - x.mean(axis=0)

# Trigger computation by explicitly calling the compute method
y = y.compute()

If you have multiple results that you want to compute at the same time, use the dask.compute function. This can share intermediate results and so be more efficient:

# compute multiple results at the same time with the compute function
min, max = dask.compute(y.min(), y.max())

Note that the compute() function returns in-memory results. It converts Dask DataFrames to Pandas DataFrames, Dask arrays to NumPy arrays, and Dask bags to lists. You should only call compute on results that will fit comfortably in memory. If your result does not fit in memory, then you might consider writing it to disk instead.

# Write larger results out to disk rather than store them in memory
my_dask_dataframe.to_parquet('myfile.parquet')
my_dask_array.to_hdf5('myfile.hdf5')
my_dask_bag.to_textfiles('myfile.*.txt')
4.6.5 Persist into Distributed Memory

Alternatively, if you are on a cluster, then you may want to trigger a computation and store the results in distributed memory. In this case you do not want to call compute, which would create a single Pandas, NumPy, or list result. Instead, you want to call persist, which returns a new Dask object that points to actively computing, or already computed results spread around your cluster’s memory.

```python
# Compute returns an in-memory non-Dask object
y = y.compute()

# Persist returns an in-memory Dask object that uses distributed storage if available
y = y.persist()
```

This is common to see after data loading and preprocessing steps, but before rapid iteration, exploration, or complex algorithms. For example, we might read in a lot of data, filter down to a more manageable subset, and then persist data into memory so that we can iterate quickly.

```python
import dask.dataframe as dd
df = dd.read_parquet('...')
df = df[df.name == 'Alice']  # select important subset of data
df = df.persist()  # trigger computation in the background

# These are all relatively fast now that the relevant data is in memory
df.groupby(df.id).balance.sum().compute()  # explore data quickly
df.groupby(df.id).balance.mean().compute()  # explore data quickly
df.id.nunique()  # explore data quickly
```

4.6.6 Lazy vs Immediate

As mentioned above, most Dask workloads are lazy, that is, they don’t start any work until you explicitly trigger them with a call to compute(). However, sometimes you do want to submit work as quickly as possible, track it over time, submit new work or cancel work depending on partial results, and so on. This can be useful when tracking or responding to real-time events, handling streaming data, or when building complex and adaptive algorithms.

For these situations, people typically turn to the futures interface which is a low-level interface like Dask delayed, but operates immediately rather than lazily.

Here is the same example with Dask delayed and Dask futures to illustrate the difference.

Delayed: Lazy

```python
@dask.delayed
def inc(x):
    return x + 1

@dask.delayed
def add(x, y):
    return x + y

a = inc(1)  # no work has happened yet
b = inc(2)  # no work has happened yet
c = add(a, b)  # no work has happened yet

c = c.compute()  # This triggers all of the above computations
```
Futures: Immediate

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

def inc(x):
    return x + 1

def add(x, y):
    return x + y

a = client.submit(inc, 1)  # work starts immediately
b = client.submit(inc, 2)  # work starts immediately
c = client.submit(add, a, b)  # work starts immediately
c = c.result()  # block until work finishes, then gather result
```

You can also trigger work with the high-level collections using the `persist` function. This will cause work to happen in the background when using the distributed scheduler.

### 4.6.7 Combining Interfaces

There are established ways to combine the interfaces above:

1. The high-level interfaces (array, bag, dataframe) have a `to_delayed` method that can convert to a sequence (or grid) of Dask delayed objects

   ```python
delays = df.to_delayed()
```

2. The high-level interfaces (array, bag, dataframe) have a `from_delayed` method that can convert from either `Delayed` or `Future` objects

   ```python
df = dd.from_delayed(delays)
df = dd.from_delayed(futures)
```

3. The `Client.compute` method converts Delayed objects into Futures

   ```python
futures = client.compute(delays)
```

4. The `dask.distributed.futures_of` function gathers futures from persisted collections

   ```python
from dask.distributed import futures_of

df = df.persist()  # start computation in the background
futures = futures_of(df)
```

5. The Dask.delayed object converts Futures into delayed objects

   ```python
delayed_value = dask.delayed(future)
```

The approaches above should suffice to convert any interface into any other. We often see some anti-patterns that do not work as well:

1. Calling low-level APIs (delayed or futures) on high-level objects (like Dask arrays or DataFrames). This downgrades those objects to their NumPy or Pandas equivalents, which may not be desired. Often people are looking for APIs like `dask.array.map_blocks` or `dask.dataframe.map_partitions` instead.
2. Calling `compute()` on Future objects. Often people want the `.result()` method instead.

3. Calling NumPy/Pandas functions on high-level Dask objects or high-level Dask functions on NumPy/Pandas objects

### 4.6.8 Conclusion

Most people who use Dask start with only one of the interfaces above but eventually learn how to use a few interfaces together. This helps them leverage the sophisticated algorithms in the high-level interfaces while also working around tricky problems with the low-level interfaces.

For more information, see the documentation for the particular user interfaces below:

- **High Level**
  - *Arrays*: Parallel NumPy
  - *Bags*: Parallel lists
  - *DataFrames*: Parallel Pandas
  - *Machine Learning*: Parallel Scikit-Learn
  - Others from external projects, like XArray

- **Low Level**
  - *Delayed*: Parallel function evaluation
  - *Futures*: Real-time parallel function evaluation

### 4.7 Array

#### 4.7.1 API

Top level user functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>all(a[, axis, out, keepdims])</code></td>
<td>Test whether all array elements along a given axis evaluate to True.</td>
</tr>
<tr>
<td><code>allclose(a, b[, rtol, atol, equal_nan])</code></td>
<td>Returns True if two arrays are element-wise equal within a tolerance.</td>
</tr>
<tr>
<td><code>angle(x[, deg])</code></td>
<td>Return the angle of the complex argument.</td>
</tr>
<tr>
<td><code>any(a[, axis, out, keepdims])</code></td>
<td>Test whether any array element along a given axis evaluates to True.</td>
</tr>
<tr>
<td><code>apply_along_axis(func1d, axis, arr, *args, ...)</code></td>
<td>Apply a function to 1-D slices along the given axis.</td>
</tr>
<tr>
<td><code>apply_over_axes(func, a, axes)</code></td>
<td>Apply a function repeatedly over multiple axes.</td>
</tr>
<tr>
<td><code>arange(*args, **kwargs)</code></td>
<td>Return evenly spaced values from <code>start</code> to <code>stop</code> with step size <code>step</code>.</td>
</tr>
<tr>
<td><code>arccos(x[, out, where, casting, order, ...])</code></td>
<td>Trigonometric inverse cosine, element-wise.</td>
</tr>
<tr>
<td><code>arccosh(x[, out, where, casting, order, ...])</code></td>
<td>Inverse hyperbolic cosine, element-wise.</td>
</tr>
<tr>
<td><code>arcsin(x[, out, where, casting, order, ...])</code></td>
<td>Inverse sine, element-wise.</td>
</tr>
<tr>
<td><code>arcsinh(x[, out, where, casting, order, ...])</code></td>
<td>Inverse hyperbolic sine element-wise.</td>
</tr>
<tr>
<td><code>arctan(x[, out, where, casting, order, ...])</code></td>
<td>Trigonometric inverse tangent, element-wise.</td>
</tr>
<tr>
<td><code>arctan2(x1, x2[, out, where, casting, order, ...])</code></td>
<td>Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arctanh(x, [l, out, where, casting, order, ...])</td>
<td>Inverse hyperbolic tangent element-wise.</td>
</tr>
<tr>
<td>argmax(a[, axis, out])</td>
<td>Returns the indices of the maximum values along an axis.</td>
</tr>
<tr>
<td>argmin(a[, axis, out])</td>
<td>Returns the indices of the minimum values along an axis.</td>
</tr>
<tr>
<td>argtopk(a, k[, axis, split_every])</td>
<td>Extract the indices of the k largest elements from a on the given axis, and return them sorted from largest to smallest.</td>
</tr>
<tr>
<td>argwhere(a)</td>
<td>Find the indices of array elements that are non-zero, grouped by element.</td>
</tr>
<tr>
<td>arround(a[, decimals, out])</td>
<td>Evenly round to the given number of decimals.</td>
</tr>
<tr>
<td>array(object[, dtype, copy, order, subok, ndmin])</td>
<td>Create an array.</td>
</tr>
<tr>
<td>asarray(a[, **kwargs])</td>
<td>Convert the input to a dask array.</td>
</tr>
<tr>
<td>atleast_1d(*arys)</td>
<td>Convert inputs to arrays with at least one dimension.</td>
</tr>
<tr>
<td>atleast_2d(*arys)</td>
<td>View inputs as arrays with at least two dimensions.</td>
</tr>
<tr>
<td>atleast_3d(*arys)</td>
<td>View inputs as arrays with at least three dimensions.</td>
</tr>
<tr>
<td>average(a[, axis, weights, returned])</td>
<td>Compute the weighted average along the specified axis.</td>
</tr>
<tr>
<td>bincount(x[, weights, min_length])</td>
<td>Count number of occurrences of each value in array of non-negative ints.</td>
</tr>
<tr>
<td>bitwise_and(x1, x2[, l, out, where, ...])</td>
<td>Compute the bit-wise AND of two arrays element-wise.</td>
</tr>
<tr>
<td>bitwise_or(x1, x2[, l, out, where, ...])</td>
<td>Compute the bit-wise OR of two arrays element-wise.</td>
</tr>
<tr>
<td>bitwise_xor(x1, x2[, l, out, where, ...])</td>
<td>Compute the bit-wise XOR of two arrays element-wise.</td>
</tr>
<tr>
<td>block(arrays[, allow_unknown_chunksizes])</td>
<td>Assemble an nd-array from nested lists of blocks.</td>
</tr>
<tr>
<td>broadcast_arrays(*args, **kwargs)</td>
<td>Broadcast any number of arrays against each other.</td>
</tr>
<tr>
<td>broadcast_to(x, shape[, chunks])</td>
<td>Broadcast an array to a new shape.</td>
</tr>
<tr>
<td>coarsen(reduction, x, axes[, trim_excess])</td>
<td>Coarsen array by applying reduction to fixed size neighborhoods</td>
</tr>
<tr>
<td>ceil(x[, l, out, where, casting, ...])</td>
<td>Return the ceiling of the input, element-wise.</td>
</tr>
<tr>
<td>choose(a, choices[, out, mode])</td>
<td>Construct an array from an index array and a set of arrays to choose from.</td>
</tr>
<tr>
<td>clip(*args, **kwargs)</td>
<td>Clip (limit) the values in an array.</td>
</tr>
<tr>
<td>compress(condition, a[, axis, out])</td>
<td>Return selected slices of an array along given axis.</td>
</tr>
<tr>
<td>concatenate(seq[, axis, ...])</td>
<td>Concatenate arrays along an existing axis.</td>
</tr>
<tr>
<td>conj(x[, out, where, casting, order, ...])</td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td>copysign(x1, x2[, l, out, where, casting, ...])</td>
<td>Change the sign of x1 to that of x2, element-wise.</td>
</tr>
<tr>
<td>corcoef(x[, y, rowvar, bias, ddof])</td>
<td>Return Pearson product-moment correlation coefficients.</td>
</tr>
<tr>
<td>cos(x[, l, out, where, casting, order, ...])</td>
<td>Cosine element-wise.</td>
</tr>
<tr>
<td>cosh(x[, l, out, where, casting, order, ...])</td>
<td>Hyperbolic cosine, element-wise.</td>
</tr>
<tr>
<td>count_nonzero(a[, axis])</td>
<td>Counts the number of non-zero values in the array a.</td>
</tr>
<tr>
<td>cov(m[, y, rowvar, bias, ddof, fweights, ...])</td>
<td>Estimate a covariance matrix, given data and weights.</td>
</tr>
<tr>
<td>cumprod(a[, axis, dtype, out])</td>
<td>Return the cumulative product of elements along a given axis.</td>
</tr>
<tr>
<td>cumsum(a[, axis, dtype, out])</td>
<td>Return the cumulative sum of the elements along a given axis.</td>
</tr>
<tr>
<td>deg2rad(x[, l, out, where, casting, order, ...])</td>
<td>Convert angles from degrees to radians.</td>
</tr>
<tr>
<td>degrees(x[, l, out, where, casting, order, ...])</td>
<td>Convert angles from radians to degrees.</td>
</tr>
<tr>
<td>diag(v[, k])</td>
<td>Extract a diagonal or construct a diagonal array.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>diagonal(a[, offset, axis1, axis2])</code></td>
<td>Return specified diagonals.</td>
</tr>
<tr>
<td><code>diff(a[, n, axis, prepend, append])</code></td>
<td>Calculate the n-th discrete difference along the given axis.</td>
</tr>
<tr>
<td><code>divmod</code></td>
<td></td>
</tr>
<tr>
<td><code>digitize(x, bins[, right])</code></td>
<td>Return the indices of the bins to which each value in input array belongs.</td>
</tr>
<tr>
<td><code>dot(a[, b[, out]])</code></td>
<td>Dot product of two arrays.</td>
</tr>
<tr>
<td><code>dstack(tup)</code></td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td><code>ediff1d(ary[, to_end, to_begin])</code></td>
<td>The differences between consecutive elements of an array.</td>
</tr>
<tr>
<td><code>einsum(subscripts, *operands[, out, dtype, . . .])</code></td>
<td>Evaluates the Einstein summation convention on the operands.</td>
</tr>
<tr>
<td><code>empty(*args, **kwargs)</code></td>
<td>Blocked variant of empty</td>
</tr>
<tr>
<td><code>empty_like(a[, dtype, chunks])</code></td>
<td>Return a new array with the same shape and type as a given array.</td>
</tr>
<tr>
<td><code>exp(x[, l, out, where, casting, order, . . .])</code></td>
<td>Calculate the exponential of all elements in the input array.</td>
</tr>
<tr>
<td><code>expm1(x[, l, out, where, casting, order, . . .])</code></td>
<td>Calculate $\exp(x) - 1$ for all elements in the array.</td>
</tr>
<tr>
<td><code>eye(N, chunks[, M, k, dtype])</code></td>
<td>Return a 2-D Array with ones on the diagonal and zeros elsewhere.</td>
</tr>
<tr>
<td><code>fabs(x[, l, out, where, casting, order, . . .])</code></td>
<td>Compute the absolute values element-wise.</td>
</tr>
<tr>
<td><code>fix(*args, **kwargs)</code></td>
<td>Round to nearest integer towards zero.</td>
</tr>
<tr>
<td><code>flatnonzero(a)</code></td>
<td>Return indices that are non-zero in the flattened version of a.</td>
</tr>
<tr>
<td><code>flip(m, axis)</code></td>
<td>Reverse element order along axis.</td>
</tr>
<tr>
<td><code>flipud(m)</code></td>
<td>Flip array in the up/down direction.</td>
</tr>
<tr>
<td><code>fliplr(m)</code></td>
<td>Flip array in the left/right direction.</td>
</tr>
<tr>
<td><code>floor(x[, l, out, where, casting, order, . . .])</code></td>
<td>Return the floor of the input, element-wise.</td>
</tr>
<tr>
<td><code>fmax(x1, x2[, l, out, where, casting, . . .])</code></td>
<td>Element-wise maximum of array elements.</td>
</tr>
<tr>
<td><code>fmin(x1, x2[, l, out, where, casting, . . .])</code></td>
<td>Element-wise minimum of array elements.</td>
</tr>
<tr>
<td><code>fmod(x1, x2[, l, out, where, casting, . . .])</code></td>
<td>Return the element-wise remainder of division.</td>
</tr>
<tr>
<td><code>frexp(x[, out1, out2], / [, out, where, . . .])</code></td>
<td>Decompose the elements of x into mantissa and twos exponent.</td>
</tr>
<tr>
<td><code>fromfunction(function, shape, **kwargs)</code></td>
<td>Construct an array by executing a function over each coordinate.</td>
</tr>
<tr>
<td><code>frompyfunc(func, nin, nout)</code></td>
<td>Takes an arbitrary Python function and returns a NumPy ufunc.</td>
</tr>
<tr>
<td><code>full(*args, **kwargs)</code></td>
<td>Blocked variant of full</td>
</tr>
<tr>
<td><code>full_like(a, fill_value[, dt, type, chunks])</code></td>
<td>Return a full array with the same shape and type as a given array.</td>
</tr>
<tr>
<td><code>gradient(f[, varargs, **kwargs])</code></td>
<td>Return the gradient of an N-dimensional array.</td>
</tr>
<tr>
<td><code>histogram(a[, bins, range, normed, weights, . . .])</code></td>
<td>Blocked variant of <code>numpy.histogram()</code>.</td>
</tr>
<tr>
<td><code>hstack(tup)</code></td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td><code>hypot(x1, x2[, l, out, where, casting, . . .])</code></td>
<td>Given the “legs” of a right triangle, return its hypotenuse.</td>
</tr>
<tr>
<td><code>imag(*args, **kwargs)</code></td>
<td>Return the imaginary part of the complex argument.</td>
</tr>
<tr>
<td><code>indices(dimensions[, dt, type, chunks])</code></td>
<td>Implements NumPy’s <code>indices</code> for Dask Arrays.</td>
</tr>
<tr>
<td><code>insert(arr, obj, values[, axis])</code></td>
<td>Insert values along the given axis before the given indices.</td>
</tr>
<tr>
<td><code>invert(x[, l, out, where, casting, order, . . .])</code></td>
<td>Compute bit-wise inversion, or bit-wise NOT, element-wise.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isclose</code></td>
<td>Returns a boolean array where two arrays are element-wise equal within a tolerance.</td>
</tr>
<tr>
<td><code>iscomplex</code></td>
<td>Returns a bool array, where True if input element is complex.</td>
</tr>
<tr>
<td><code>isfinite</code></td>
<td>Test element-wise for finiteness (not infinity or not Not a Number).</td>
</tr>
<tr>
<td><code>isin</code></td>
<td>Calculates element in test_elements, broadcasting over element only.</td>
</tr>
<tr>
<td><code>isinf</code></td>
<td>Test element-wise for positive or negative infinity.</td>
</tr>
<tr>
<td><code>isnan</code></td>
<td>Test element-wise for NaN and return result as a boolean array.</td>
</tr>
<tr>
<td><code>isnull</code></td>
<td>pandas.isnull for dask arrays.</td>
</tr>
<tr>
<td><code>isposinf</code></td>
<td>Test element-wise for positive infinity, return result as bool array.</td>
</tr>
<tr>
<td><code>isreal</code></td>
<td>Test element-wise for positive infinity, return result as bool array.</td>
</tr>
<tr>
<td><code>ldexp</code></td>
<td>Returns x1 * 2**x2, element-wise.</td>
</tr>
<tr>
<td><code>linspace</code></td>
<td>Return num evenly spaced values over the closed interval [start, stop].</td>
</tr>
<tr>
<td><code>log</code></td>
<td>Natural logarithm, element-wise.</td>
</tr>
<tr>
<td><code>log10</code></td>
<td>Return the base 10 logarithm of the input array, element-wise.</td>
</tr>
<tr>
<td><code>log1p</code></td>
<td>Return the natural logarithm of one plus the input array, element-wise.</td>
</tr>
<tr>
<td><code>log2</code></td>
<td>Base-2 logarithm of x.</td>
</tr>
<tr>
<td><code>logaddexp</code></td>
<td>Logarithm of the sum of exponentiations of the inputs.</td>
</tr>
<tr>
<td><code>logaddexp2</code></td>
<td>Logarithm of the sum of exponentiations of the inputs in base-2.</td>
</tr>
<tr>
<td><code>logical_and</code></td>
<td>Compute the truth value of x1 AND x2 element-wise.</td>
</tr>
<tr>
<td><code>logical_not</code></td>
<td>Compute the truth value of NOT x element-wise.</td>
</tr>
<tr>
<td><code>logical_or</code></td>
<td>Compute the truth value of x1 OR x2 element-wise.</td>
</tr>
<tr>
<td><code>logical_xor</code></td>
<td>Compute the truth value of x1 XOR x2, element-wise.</td>
</tr>
<tr>
<td><code>map_blocks</code></td>
<td>Map a function across all blocks of a dask array.</td>
</tr>
<tr>
<td><code>map_overlap</code></td>
<td>Map a function over blocks of the array with some overlap</td>
</tr>
<tr>
<td><code>matmul</code></td>
<td>Return the maximum of an array or maximum along an axis.</td>
</tr>
<tr>
<td><code>maximum</code></td>
<td>Element-wise maximum of array elements.</td>
</tr>
<tr>
<td><code>mean</code></td>
<td>Compute the arithmetic mean along the specified axis.</td>
</tr>
<tr>
<td><code>meshgrid</code></td>
<td>Return coordinate matrices from coordinate vectors.</td>
</tr>
<tr>
<td><code>minimum</code></td>
<td>Return the minimum of an array or minimum along an axis.</td>
</tr>
<tr>
<td><code>modf</code></td>
<td>Return the fractional and integral parts of an array, element-wise.</td>
</tr>
<tr>
<td><code>moment</code></td>
<td>Compute the moment of the array.</td>
</tr>
<tr>
<td><code>nanargmax</code></td>
<td>Compute the index of the maximum value ignoring NaNs.</td>
</tr>
<tr>
<td><code>nanargmin</code></td>
<td>Compute the index of the minimum value ignoring NaNs.</td>
</tr>
</tbody>
</table>
Table 4 – continued from previous page

- `nancumprod(a[, axis, dtype, out])`: Return the cumulative product of array elements over a given axis treating Not a Numbers (NaNs) as one.
- `nancumsum(a[, axis, dtype, out])`: Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
- `nanmax(a[, axis, out, keepdims])`: Return the maximum of an array or maximum along an axis, ignoring any NaNs.
- `nanmean(a[, axis, dtype, out, keepdims])`: Compute the arithmetic mean along the specified axis, ignoring NaNs.
- `nanmin(a[, axis, out, keepdims])`: Return minimum of an array or minimum along an axis, ignoring any NaNs.
- `nanprod(a[, axis, dtype, out, keepdims])`: Return the product of array elements over a given axis treating Not a Numbers (NaNs) as ones.
- `nanstd(a[, axis, dtype, out, ddof, keepdims])`: Compute the standard deviation along the specified axis, while ignoring NaNs.
- `nansum(a[, axis, dtype, out, keepdims])`: Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
- `nanvar(a[, axis, dtype, out, ddof, keepdims])`: Compute the variance along the specified axis, while ignoring NaNs.
- `nan_to_num(*args, **kwargs)`: Replace NaN with zero and infinity with large finite numbers.
- `nextafter(x1, x2[, out, where, casting, ...])`: Return the next floating-point value after x1 towards x2, element-wise.
- `nonzero(a)`: Return the indices of the elements that are non-zero.
- `notnull(values)`: pandas.notnull for dask arrays.
- `ones(*args, **kwargs)`: Blocked variant of ones.
- `ones_like(a[, dtype, chunks])`: Return an array of ones with the same shape and type as a given array.
- `outer(a, b[, out])`: Compute the outer product of two vectors.
- `pad(array, pad_width, mode, **kwargs)`: Pads an array.
- `percentile(a, q[, interpolation, method])`: Approximate percentile of 1-D array.
- `PerformanceWarning`: A warning given when bad chunking may cause poor performance.
- `piecewise(x, condlist, funclist, *args, **kw)`: Evaluate a piecewise-defined function.
- `prod(a[, axis, dtype, out, keepdims, initial])`: Return the product of array elements over a given axis.
- `ptp(a[, axis, out, keepdims])`: Range of values (maximum - minimum) along an axis.
- `rad2deg(x[, l, out, where, casting, order, ...])`: Convert angles from radians to degrees.
- `radians(x[, l, out, where, casting, order, ...])`: Convert angles from degrees to radians.
- `ravel(a[, order])`: Return a contiguous flattened array.
- `real(*args, **kwargs)`: Return the real part of the complex argument.
- `rechunk(x, chunks[, threshold, block_size_limit])`: Convert blocks in dask array x for new chunks.
- `repeat(a, repeats[, axis])`: Repeat elements of an array.
- `reshape(x, shape)`: Reshape array to new shape.
- `result_type(*arrays_and_dtypes)`: Returns the type that results from applying the NumPy type promotion rules to the arguments.
- `rint(x[, l, out, where, casting, order, ...])`: Round elements of the array to the nearest integer.
- `roll(a, shift[, axis])`: Roll array elements along a given axis.
- `round(a[, decimals, out])`: Round an array to the given number of decimals.
- `sign(x[, l, out, where, casting, order, ...])`: Returns an element-wise indication of the sign of a number.
- `signbit(x[, l, out, where, casting, order, ...])`: Returns element-wise True where signbit is set (less than zero).

4.7. Array 57

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sin(x, l[, \text{out, where, casting, order, ...}]))</td>
<td>Trigonometric sine, element-wise.</td>
</tr>
<tr>
<td>(\sinh(x, l[, \text{out, where, casting, order, ...}]))</td>
<td>Hyperbolic sine, element-wise.</td>
</tr>
<tr>
<td>(\sqrt{x, l[, \text{out, where, casting, order, ...}]})</td>
<td>Return the non-negative square-root of an array, element-wise.</td>
</tr>
<tr>
<td>(\text{square}(x, l[, \text{out, where, casting, order, ...}]))</td>
<td>Return the element-wise square of the input.</td>
</tr>
<tr>
<td>(\text{squeeze}(a[, \text{axis}]))</td>
<td>Remove single-dimensional entries from the shape of an array.</td>
</tr>
<tr>
<td>(\text{stack}(seq[, \text{axis}]))</td>
<td>Stack arrays along a new axis</td>
</tr>
<tr>
<td>(\text{std}(a[, \text{axis, dtype, out, ddof, keepdims}]))</td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td>(\text{sum}(a[, \text{axis, dtype, out, keepdims, initial}]))</td>
<td>Sum of array elements over a given axis.</td>
</tr>
<tr>
<td>(\text{take}(a, \text{indices[, axis, out, mode]}))</td>
<td>Take elements from an array along an axis.</td>
</tr>
<tr>
<td>(\tan(x, l[, \text{out, where, casting, order, ...}]))</td>
<td>Compute tangent element-wise.</td>
</tr>
<tr>
<td>(\text{tanh}(x, l[, \text{out, where, casting, order, ...}]))</td>
<td>Compute hyperbolic tangent element-wise.</td>
</tr>
<tr>
<td>(\text{tensordot}(a, b[, \text{axes}]))</td>
<td>Compute tensor dot product along specified axes for arrays &gt;= 1-D.</td>
</tr>
<tr>
<td>(\text{tile}(A, \text{reps}))</td>
<td>Construct an array by repeating (A) the number of times given by (\text{reps}).</td>
</tr>
<tr>
<td>(\text{tril}(m[, k]))</td>
<td>Lower triangle of an array with elements above the (k)-th diagonal zeroed.</td>
</tr>
<tr>
<td>(\text{triu}(m[, k]))</td>
<td>Upper triangle of an array with elements above the (k)-th diagonal zeroed.</td>
</tr>
<tr>
<td>(\text{trunc}(x, l[, \text{out, where, casting, order, ...}]))</td>
<td>Return the truncated value of the input, element-wise.</td>
</tr>
<tr>
<td>(\text{unique}(ar[, \text{return_index, return_inverse, ...}]))</td>
<td>Find the unique elements of an array.</td>
</tr>
<tr>
<td>(\text{unravel_index}(\text{indices, shape[, order]}))</td>
<td>Converts a flat index or array of flat indices into a tuple of coordinate arrays.</td>
</tr>
<tr>
<td>(\text{var}(a[, \text{axis, dtype, out, ddof, keepdims}]))</td>
<td>Compute the variance along the specified axis.</td>
</tr>
<tr>
<td>(\text{vdot}(a, b))</td>
<td>Return the dot product of two vectors.</td>
</tr>
<tr>
<td>(\text{vstack}(\text{tup}))</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
<tr>
<td>(\text{where}(\text{condition[, x, y]}))</td>
<td>Return elements chosen from (x) or (y) depending on (\text{condition}).</td>
</tr>
<tr>
<td>(\text{zeros}(*\text{args, **kwargs}))</td>
<td>Blocked variant of (\text{zeros})</td>
</tr>
<tr>
<td>(\text{zeros_like}(a[, \text{dtype, chunks}]))</td>
<td>Return an array of zeros with the same shape and type as a given array.</td>
</tr>
</tbody>
</table>

### Fast Fourier Transforms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{fft.fft_wrap}(\text{fft_func[, kind, dtype]}))</td>
<td>Wrap 1D, 2D, and ND real and complex FFT functions</td>
</tr>
<tr>
<td>(\text{fft.fft}(a[, n, axis]))</td>
<td>Wrapping of numpy.fft.fft</td>
</tr>
<tr>
<td>(\text{fft.fft2}(a[, s, axes]))</td>
<td>Wrapping of numpy.fft.fft2</td>
</tr>
<tr>
<td>(\text{fft.fftn}(a[, s, axes]))</td>
<td>Wrapping of numpy.fft.fftn</td>
</tr>
<tr>
<td>(\text{fft.ifft}(a[, n, axis]))</td>
<td>Wrapping of numpy.fft.ifft</td>
</tr>
<tr>
<td>(\text{fft.ifft2}(a[, s, axes]))</td>
<td>Wrapping of numpy.fft.ifft2</td>
</tr>
<tr>
<td>(\text{fft.ifftn}(a[, s, axes]))</td>
<td>Wrapping of numpy.fft.ifftn</td>
</tr>
<tr>
<td>(\text{fft.rfft}(a[, n, axis]))</td>
<td>Wrapping of numpy.fft.rfft</td>
</tr>
<tr>
<td>(\text{fft.rfft2}(a[, s, axes]))</td>
<td>Wrapping of numpy.fft.rfft2</td>
</tr>
</tbody>
</table>

Continued on next page
Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fft.rfftn</code></td>
<td>Wrapping of <code>numpy.fft.rfftn</code></td>
</tr>
<tr>
<td><code>fft.irfft</code></td>
<td>Wrapping of <code>numpy.fft.irfft</code></td>
</tr>
<tr>
<td><code>fft.irfftn</code></td>
<td>Wrapping of <code>numpy.fft.irfftn</code></td>
</tr>
<tr>
<td><code>fft.hfft</code></td>
<td>Wrapping of <code>numpy.fft.hfft</code></td>
</tr>
<tr>
<td><code>fft.ihfft</code></td>
<td>Wrapping of <code>numpy.fft.ihfft</code></td>
</tr>
<tr>
<td><code>fft.fftfreq</code></td>
<td>Return the Discrete Fourier Transform sample frequencies.</td>
</tr>
<tr>
<td><code>fft.rfftfreq</code></td>
<td>Return the Discrete Fourier Transform sample frequencies (for usage with <code>rfft</code>, <code>irfft</code>).</td>
</tr>
<tr>
<td><code>fft.fftshift</code></td>
<td>Shift the zero-frequency component to the center of the spectrum.</td>
</tr>
<tr>
<td><code>fft.ifftshift</code></td>
<td>The inverse of <code>fftshift</code>.</td>
</tr>
</tbody>
</table>

Linear Algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>linalg.cholesky</code></td>
<td>Returns the Cholesky decomposition, ( A = LL^* ) or ( A = U^*U ) of a Hermitian positive-definite matrix ( A ).</td>
</tr>
<tr>
<td><code>linalg.inv</code></td>
<td>Compute the inverse of a matrix with LU decomposition and forward / backward substitutions.</td>
</tr>
<tr>
<td><code>linalg.lstsq</code></td>
<td>Return the least-squares solution to a linear matrix equation using QR decomposition.</td>
</tr>
<tr>
<td><code>linalg.lu</code></td>
<td>Compute the lu decomposition of a matrix.</td>
</tr>
<tr>
<td><code>linalg.norm</code></td>
<td>Matrix or vector norm.</td>
</tr>
<tr>
<td><code>linalg.qr</code></td>
<td>Compute the qr factorization of a matrix.</td>
</tr>
<tr>
<td><code>linalg.solve</code></td>
<td>Solve the equation ( a \ x = b ) for ( x ).</td>
</tr>
<tr>
<td><code>linalg.solve_triangular</code></td>
<td>Solve the equation ( a \ x = b ) for ( x ), assuming ( a ) is a triangular matrix.</td>
</tr>
<tr>
<td><code>linalg.svd</code></td>
<td>Compute the singular value decomposition of a matrix.</td>
</tr>
<tr>
<td><code>linalg.svd_compressed</code></td>
<td>Randomly compressed rank-k thin Singular Value Decomposition.</td>
</tr>
<tr>
<td><code>linalg.sqf</code></td>
<td>Direct Short-and-Fat QR</td>
</tr>
<tr>
<td><code>linalg.tsqf</code></td>
<td>Direct Tall-and-Skinny QR</td>
</tr>
</tbody>
</table>

Masked Arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.average</code></td>
<td>Return the weighted average of array over the given axis.</td>
</tr>
<tr>
<td><code>ma.filled</code></td>
<td>Return input as an array with masked data replaced by a fill value.</td>
</tr>
<tr>
<td><code>ma.fix_invalid</code></td>
<td>Return input with invalid data masked and replaced by a fill value.</td>
</tr>
<tr>
<td><code>ma.getdata</code></td>
<td>Return the data of a masked array as an ndarray.</td>
</tr>
<tr>
<td><code>ma.getmaskarray</code></td>
<td>Return the mask of a masked array, or full boolean array of False.</td>
</tr>
<tr>
<td><code>ma.masked_array</code></td>
<td>An array class with possibly masked values.</td>
</tr>
<tr>
<td><code>ma.masked_equal</code></td>
<td>Mask an array where equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater</code></td>
<td>Mask an array where greater than a given value.</td>
</tr>
</tbody>
</table>
### Table 7 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.masked_greater_equal(x, value[, copy])</code></td>
<td>Mask an array where greater than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_inside(x, v1, v2[, copy])</code></td>
<td>Mask an array inside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_invalid(a[, copy])</code></td>
<td>Mask an array where invalid values occur (NaNs or infs).</td>
</tr>
<tr>
<td><code>ma.masked_less(x, value[, copy])</code></td>
<td>Mask an array where less than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_less_equal(x, value[, copy])</code></td>
<td>Mask an array where less than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_not_equal(x, value[, copy])</code></td>
<td>Mask an array where not equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_outside(x, v1, v2[, copy])</code></td>
<td>Mask an array outside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_values(x, value[, rtol, atol, ...])</code></td>
<td>Mask using floating point equality.</td>
</tr>
<tr>
<td><code>ma.masked_where(condition, a[, copy])</code></td>
<td>Mask an array where a condition is met.</td>
</tr>
<tr>
<td><code>ma.set_fill_value(a, fill_value)</code></td>
<td>Set the filling value of a, if a is a masked array.</td>
</tr>
</tbody>
</table>

### Random

- `random.beta(a, b[, size])` Draw samples from a Beta distribution.
- `random.binomial(n, p[, size])` Draw samples from a binomial distribution.
- `random.chisquare(df[, size])` Draw samples from a chi-square distribution.
- `random.choice(a[, size, replace, p])` Generates a random sample from a given 1-D array.
- `random.exponential([scale, size])` Draw samples from an exponential distribution.
- `random.f(dfnum, dfden[, size])` Draw samples from an F distribution.
- `random.gamma(shape[, scale, size])` Draw samples from a Gamma distribution.
- `random.geometric(p[, size])` Draw samples from the geometric distribution.
- `random.gumbel([loc, scale, size])` Draw samples from a Gumbel distribution.
- `random.hypergeometric(ngood, nbad, nsample)` Draw samples from a Hypergeometric distribution.
- `random.laplace([loc, scale, size])` Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).
- `random.logistic([loc, scale, size])` Draw samples from a logistic distribution.
- `random.lognormal([mean, sigma, size])` Draw samples from a log-normal distribution.
- `random.logseries(p[, size])` Draw samples from a logarithmic series distribution.
- `random.negative_binomial(n, p[, size])` Draw samples from a negative binomial distribution.
- `random.noncentral_chisquare(df, nonc[, size])` Draw samples from a noncentral chi-square distribution.
- `random.noncentral_f(dfnum, dfden, nonc[, size])` Draw samples from the noncentral F distribution.
- `random.normal([loc, scale, size])` Draw random samples from a normal (Gaussian) distribution.
- `random.pareto(a[, size])` Draw samples from a Pareto II or Lomax distribution with specified shape.
- `random.poisson([lam, size])` Draw samples from a Poisson distribution.
- `random.power(a[, size])` Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
- `random.randint(low[, high, size, dtype])` Return random integers from low (inclusive) to high (exclusive).
- `random.random([size])` Return random floats in the half-open interval [0.0, 1.0).
- `random.random_sample([size])` Return random floats in the half-open interval [0.0, 1.0).
- `random.rayleigh([scale, size])` Draw samples from a Rayleigh distribution.
- `random.standard_cauchy([size])` Draw samples from a standard Cauchy distribution with mode = 0.
Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>random.standard_exponential([size])</code></td>
<td>Draw samples from the standard exponential distribution.</td>
</tr>
<tr>
<td><code>random.standard_gamma(\text{shape[, size]} \right)</code></td>
<td>Draw samples from a standard Gamma distribution.</td>
</tr>
<tr>
<td><code>random.standard_normal([size])</code></td>
<td>Draw samples from a standard Normal distribution (mean=0, stdev=1).</td>
</tr>
<tr>
<td><code>random.standard_t(df[, size])</code></td>
<td>Draw samples from a standard Student's ( t ) distribution with ( df ) degrees of freedom.</td>
</tr>
<tr>
<td><code>random.triangular(left, mode, right[, size])</code></td>
<td>Draw samples from the triangular distribution over the interval ([left, right]).</td>
</tr>
<tr>
<td><code>random.uniform([low, high, size])</code></td>
<td>Draw samples from a uniform distribution.</td>
</tr>
<tr>
<td><code>random.vonmises(mu, kappa[, size])</code></td>
<td>Draw samples from a von Mises distribution.</td>
</tr>
<tr>
<td><code>random.wald(mean, scale[, size])</code></td>
<td>Draw samples from a Wald, or inverse Gaussian, distribution.</td>
</tr>
<tr>
<td><code>random.weibull(a[, size])</code></td>
<td>Draw samples from a Weibull distribution.</td>
</tr>
<tr>
<td><code>random.zipf(a[, size])</code></td>
<td>Standard distributions</td>
</tr>
</tbody>
</table>

**Stats**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stats.ttest_ind(a, b[, axis, equal_var])</code></td>
<td>Calculate the ( T )-test for the means of two independent samples of scores.</td>
</tr>
<tr>
<td><code>stats.ttest_1samp(a, popmean[, axis, nan_policy])</code></td>
<td>Calculate the ( T )-test for the mean of ONE group of scores.</td>
</tr>
<tr>
<td><code>stats.ttest_rel(a, b[, axis, nan_policy])</code></td>
<td>Calculate the ( T )-test on TWO RELATED samples of scores, ( a ) and ( b ).</td>
</tr>
<tr>
<td><code>stats.chisquare(f_obs[, f_exp, ddof, axis])</code></td>
<td>Calculate a one-way chi square test.</td>
</tr>
<tr>
<td><code>stats.power_divergence(f_obs[, f_exp, ddof, \ldots])</code></td>
<td>Cressie-Read power divergence statistic and goodness of fit test.</td>
</tr>
<tr>
<td><code>stats.skew(a[, axis, bias, nan_policy])</code></td>
<td>Compute the skewness of a data set.</td>
</tr>
<tr>
<td><code>stats.skewtest(a[, axis, nan_policy])</code></td>
<td>Test whether the skew is different from the normal distribution.</td>
</tr>
<tr>
<td><code>stats.kurtosis(a[, axis, fisher, bias, \ldots])</code></td>
<td>Compute the kurtosis (Fisher or Pearson) of a dataset.</td>
</tr>
<tr>
<td><code>stats.kurtosistest(a[, axis, nan_policy])</code></td>
<td>Test whether a dataset has normal kurtosis.</td>
</tr>
<tr>
<td><code>stats.normaltest(a[, axis, nan_policy])</code></td>
<td>Test whether a sample differs from a normal distribution.</td>
</tr>
<tr>
<td><code>stats.f_oneway(*args)</code></td>
<td>Performs a 1-way ANOVA.</td>
</tr>
<tr>
<td><code>stats.moment(a[, moment, axis, nan_policy])</code></td>
<td>Calculate the nth moment about the mean for a sample.</td>
</tr>
</tbody>
</table>

**Image Support**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>image.imread(filename[, imread, preprocess])</code></td>
<td>Read a stack of images into a dask array</td>
</tr>
</tbody>
</table>

**Slightly Overlapping Computations**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>overlap.overlap(x, depth, boundary)</code></td>
<td>Share boundaries between neighboring blocks</td>
</tr>
<tr>
<td><code>overlap.map_overlap(x, func, depth[, \ldots])</code></td>
<td>Map a function over blocks of the array with some overlap</td>
</tr>
<tr>
<td><code>overlap.trim_internal(x, axes[, boundary])</code></td>
<td>Trim sides from each block</td>
</tr>
<tr>
<td><code>overlap.trim_overlap(x, depth[, boundary])</code></td>
<td>Trim sides from each block</td>
</tr>
</tbody>
</table>
Create and Store Arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>from_array(x[, chunks, name, lock, asarray, ...])</code></td>
<td>Create dask array from something that looks like an array. Input must have a <code>.shape</code> and support numpy-style slicing.</td>
</tr>
<tr>
<td><code>from_delayed(value, shape, dtype[, name])</code></td>
<td>Create a dask array from a dask delayed value.</td>
</tr>
<tr>
<td><code>from_npy_stack(dirname[, mmap_mode])</code></td>
<td>Load dask array from stack of npy files.</td>
</tr>
<tr>
<td><code>from_zarr(url[, component, storage_options, ...])</code></td>
<td>Load array from the zarr storage format.</td>
</tr>
<tr>
<td><code>from_tiledb(uri[, attribute, chunks, ...])</code></td>
<td>Load array from the TileDB storage format.</td>
</tr>
<tr>
<td><code>store(sources, targets[, lock, regions, ...])</code></td>
<td>Store dask arrays in array-like objects, overwrite data in target.</td>
</tr>
<tr>
<td><code>to_hdf5(filename, *args, **kwargs)</code></td>
<td>Store arrays in HDF5 file.</td>
</tr>
<tr>
<td><code>to_zarr(arr, url[, component, ...])</code></td>
<td>Save array to the zarr storage format.</td>
</tr>
<tr>
<td><code>to_npy_stack(dirname, x[, axis])</code></td>
<td>Write dask array to a stack of .npy files.</td>
</tr>
<tr>
<td><code>to_tiledb(darray, uri[, compute, ...])</code></td>
<td>Save array to the TileDB storage format.</td>
</tr>
</tbody>
</table>

Generalized Ufuncs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply_gufunc(func, signature, *args, **kwargs)</code></td>
<td>Apply a generalized ufunc or similar python function to arrays.</td>
</tr>
<tr>
<td><code>as_gufunc([signature])</code></td>
<td>Decorator for <code>dask.array.gufunc</code>.</td>
</tr>
<tr>
<td><code>gufunc(pyfunc, **kwargs)</code></td>
<td>Binds <code>pyfunc</code> into <code>dask.array.apply_gufunc</code> when called.</td>
</tr>
</tbody>
</table>

Internal functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>blockwise(func, out_ind, *args, **kwargs)</code></td>
<td>Tensor operation: Generalized inner and outer products.</td>
</tr>
<tr>
<td><code>normalize_chunks(chunks[, shape, limit, ...])</code></td>
<td>Normalize chunks to tuple of tuples.</td>
</tr>
</tbody>
</table>

Other functions

```python
dask.array.from_array(x, chunks='auto', 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 block-size like 1000. - A blockshape like (1000, 1000). - Explicit sizes of all blocks along all dimensions like
    - ((1000, 1000, 500), (400, 400)).
  - A size in bytes, like “100 MiB” which will choose a uniform block-like shape
  - The word “auto” which acts like the above, but uses a configuration value `array.chunk-size` for the chunk size
  - -1 or None as a blocksize indicate the size of the corresponding dimension.
name [str, optional] The key name to use for the array. Defaults to a hash of \( x \). By default, hash uses python’s standard sha1. This behaviour can be changed by installing cityhash, xxhash or murmurhash. If installed, a large-factor speedup can be obtained in the tokenisation step. 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 \( x \) doesn’t support fancy indexing (e.g. indexing with lists or arrays) then set to False. False is True.

Examples

```python
>>> x = h5py.File('...')['/data/path']  # doctest: +SKIP
>>> a = da.from_array(x, chunks=(1000, 1000))  # doctest: +SKIP
```

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.

```python
>>> a = da.from_array(x, chunks=(1000, 1000), lock=True)  # doctest: +SKIP
```

If your underlying datastore has a .chunks attribute (as h5py and zarr datasets do) then a multiple of that chunk shape will be used if you do not provide a chunk shape.

```python
>>> a = da.from_array(x, chunks='auto')  # doctest: +SKIP
>>> a = da.from_array(x, chunks='100 MiB')  # doctest: +SKIP
>>> a = da.from_array(x)  # doctest: +SKIP
```

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

```python
>>> from dask import delayed
>>> value = delayed(np.ones)(5)
>>> array = from_delayed(value, (5,), float)
>>> array
```

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 Delayed or iterable of array-likes and/or Delayeds
  - 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 list of tuples 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. If this is a tuple, the contents will be assumed to be slices, so do not provide a tuple of tuples.
- **compute**: boolean, optional
  - If true compute immediately, return `dask.delayed.Delayed` otherwise
- **return_stored**: boolean, optional
  - Optionally return the stored result (default False).

**Examples**

```python
>>> x = ...  # doctest: +SKIP

>>> import h5py  # doctest: +SKIP
>>> f = h5py.File('myfile.hdf5')  # doctest: +SKIP
>>> dset = f.create_dataset('/data', shape=x.shape, ...
                      chunks=x.chunks,
                      dtype='f8')  # doctest: +SKIP

>>> store(x, dset)  # doctest: +SKIP

Alternatively store many arrays at the same time

```python
>>> store([x, y, z], [dset1, dset2, dset3])  # doctest: +SKIP
```

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**

```python
>>> 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

```python
>>> x = np.arange(24).reshape((4, 6))
```
```
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]])
```

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

You must avoid excess elements explicitly

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

```python
>>> 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

```python
>>> 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)
```

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

```python
>>> 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

```python
>>> 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=<no value>)

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 input array.
If the default value is passed, then `keepdims` will not be passed through to the `all` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class’ method does not implement `keepdims` any exceptions will be raised.

**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**

```python
def np.all([[True, False], [True, True]])
False
```

```python
def np.all([[True, False], [True, True]], axis=0)
array([True, False])
```

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

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

```python
def o=np.array([False])
def z=np.all([-1, 4, 5], out=o)
def id(z), id(o), z
# doctest: +SKIP
(28293632, 28293632, array([True]))
```

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, equal

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

\[ \text{absolute}(a - b) <= (\text{atol} + \text{rtol} \times \text{absolute}(b)) \]

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

The comparison of \(a\) and \(b\) uses standard broadcasting, which means that \(a\) and \(b\) need not have the same shape in order for \(\text{allclose}(a, b)\) to evaluate to True. The same is true for \(\text{equal}\) but not \(\text{array_equal}\).

Examples

```python
>>> 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.

..versionchanged:: 1.16.0 This function works on subclasses of ndarray like ma.array.

See also:
arctan2, absolute

Examples
>>> np.angle([1.0, 1.0j, 1+1j])  # in radians  # doctest: +SKIP
array([ 0. , 1.57079633, 0.78539816])

>>> np.angle(1+1j, deg=True)  # in degrees  # doctest: +SKIP
45.0

dask.array.any(a, axis=None, out=None, keepdims=<no value>)
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 (\(\text{axis} = \text{None}\)) 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 input array.

  If the default value is passed, then keepdims will not be passed through to the any method of sub-classes of ndarray, however any non-default value will be. If the sub-class’ method does not implement keepdims any exceptions will be raised.

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 \(\text{True}\) because these are not equal to zero.

Examples

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

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

>>> # Check now that z is a reference to o
>>> z is o
True
>>> id(z), id(o) # identity of z and o # doctest: +SKIP
(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`.

This is equivalent to (but faster than) the following use of `ndindex` and `s_`, which sets each of `ii`, `jj`, and `kk` to a tuple of indices:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        f = func1d(arr[ii + s_[:,] + kk])
        Nj = f.shape
        for jj in ndindex(Nj):
            out[ii + jj + kk] = f[jj]
```

Equivalently, eliminating the inner loop, this can be expressed as:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        out[ii + s_[...,] + kk] = func1d(arr[ii + s_[:,] + kk])
```

**Parameters**

- `func1d` [function (M,) -> (Nj,...)] 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 (Ni,..., M, Nk,...)] Input array.
- `args` [any] Additional arguments to `func1d`.
- `kwargs` [any] Additional named arguments to `func1d`.

**New in version 1.9.0.**

**Returns**

- `out` [ndarray (Ni,..., Nj,..., Nk,...)] The output array. The shape of `out` is identical to the shape of `arr`, except along the `axis` dimension. This axis is removed, and replaced with new dimensions equal to the shape of the return value of `func1d`. So if `func1d` returns a scalar `out` will have one fewer dimensions than `arr`.  

70 Chapter 4. Index
See also:

**apply_over_axes** Apply a function repeatedly over multiple axes.

**Examples**

```python
def my_func(a):
    ...  #"Average first and last element of a 1-D array"
    ...  return (a[0] + a[-1]) / 2

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 returns a 1D array, the number of dimensions in `outarr` is the same as `arr`.

```python
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]])
```

For a function that returns a higher dimensional array, those dimensions are inserted in place of the `axis` dimension.

```python
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
np.apply_along_axis(np.diag, -1, b)
array([[[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]],
       [[4, 0, 0],
       [0, 5, 0],
       [0, 0, 6]],
       [[7, 0, 0],
       [0, 8, 0],
       [0, 0, 9]]])
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**

```python
>>> 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:

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

Tuple axis arguments to ufuncs are equivalent:

```python
>>> 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`.
- `dtype` [numpy.dtype] Output dtype. Omit to infer it from start, stop, step

**Returns**
samples [dask array]

See also:

dask.array.linspace
dask.array.arccos(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature, extobj)

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, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **angle** [ndarray] The angle of the ray intersecting the unit circle at the given x-coordinate in radians [0, pi]. This is a scalar if x is a scalar.

See also:

cos, arctan, arcsin, emath.arccos

Notes

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

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

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

The inverse \( \cos \) is also known as \( \acos \) or \( \cos^{-1} \).

References


Examples

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

```python
>>> np.arccos([1, -1]) # doctest: +SKIP
array([ 0.0, 3.14159265])
```
Plot arccos:

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

```
dask.array.arccosh(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature, extobj)
```

Inverse hyperbolic cosine, element-wise.

**Parameters**

- `x` [array_like] Input array.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**Returns**

- `arccosh` [ndarray] Array of the same shape as `x`. This is a scalar if `x` is a scalar.

**See also:**

cosh, arcsinh, sinh, arctanh, tanh

**Notes**

`arccosh` is a multivalued function: for each `x` there are infinitely many numbers `z` such that \( \cosh(z) = x \). The convention is to return the `z` whose imaginary part lies in \([-\pi, \pi]\) and the real part in \([0, \infty]\).

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

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

**References**

[1], [2]

**Examples**

```python
>>> np.arccosh([np.e, 10.0])  # doctest: +SKIP
array([1.65745445, 2.99322285])
```

```python
>>> np.arccosh(1)  # doctest: +SKIP
0.0
```
dask.array.arcsin(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True)
Inverse sine, element-wise.

Parameters

x [array_like] y-coordinate on the unit circle.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

angle [ndarray] The inverse sine of each element in x, in radians and in the closed interval [-pi/2, pi/2]. This is a scalar if x is a scalar.

See also:

sin, cos, arccos, tan, arctan, arctan2, emath.arcsin

Notes

arcsin is a multivalued function: for each x there are infinitely many numbers z such that sin(z) = x. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

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

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

The inverse sine is also known as asin or sin⁻¹.

References


Examples

```python
>>> np.arcsin(1)  # pi/2 # doctest: +SKIP
1.5707963267948966
>>> np.arcsin(-1) # -pi/2 # doctest: +SKIP
-1.5707963267948966
>>> np.arcsin(0)  # doctest: +SKIP
0.0
```

dask.array.arcsinh(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True)
Inverse hyperbolic sine element-wise.

Parameters
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, \( \text{arcsinh} \) always returns real output. For each value that cannot be expressed as a real number or infinity, it returns \( \text{nan} \) and sets the \text{invalid} floating point error flag.

For complex-valued input, \( \text{arccosh} \) is a complex analytical function that has branch cuts \([1j, \infty j]\) and \([-1j, -\infty j]\) and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as \( \text{asinh} \) or \( \sinh^{-1} \).

**References**

[1], [2]

**Examples**

```python
>>> np.arcsinh(np.array([np.e, 10.0]))  # doctest: +SKIP
array([ 1.72538256, 2.99822295])
```

dask.array.arrctan(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Trigonometric inverse tangent, element-wise.

The inverse of \( \tan \), so that if \( y = \tan(x) \) then \( x = \text{arctan}(y) \).

**Parameters**

- \( x \) [array_like]

**Returns**
out [ndarray or scalar] Out has the same shape as x. Its real part is in \([-\pi/2, \pi/2]\)
(arctan(+/-\infty) returns +/-\pi/2). This is a scalar if x is a scalar.

See also:

arctan2 The “four quadrant” arctan of the angle formed by (x, y) and the positive x-axis.

angle Argument of complex values.

Notes

arctan is a multi-valued function: for each x there are infinitely many numbers z such that tan(z) = x. The
convention is to return the angle z whose real part lies in [-\pi/2, \pi/2].

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

For complex-valued input, arctan is a complex analytic function that has \([ij, \infty j]\) and \([-ij, -\infty j]\) 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,

Examples

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

```python
>>> np.arctan([0, 1])  # doctest: +SKIP
array([ 0. , 0.78539816])

>>> np.pi/4  # doctest: +SKIP
0.78539816339744828
```

Plot arctan:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> x = np.linspace(-10, 10)  # doctest: +SKIP
>>> plt.plot(x, np.arctan(x))  # doctest: +SKIP
>>> plt.axis('tight')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

dask.array.arctan2(x1, x2, *, out=None, where=True, casting='same_kind', order='K',
dtype=None, subok=True, signature=None, extobj=None)

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 =
+/-\infty (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.

- **out**: [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

- **where**: [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs: For other keyword-only arguments, see the ufunc docs.

Returns

- **angle**: [ndarray] Array of angles in radians, in the range [-pi, pi]. This is a scalar if both x1 and x2 are scalars.

See also:

arctan, tan, angle

Notes

*arctan2* is identical to the *atan2* function of the underlying C library. The following special values are defined in the C standard: [1]

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>arctan2(x1,x2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-0</td>
<td>+0</td>
<td>+/- 0</td>
</tr>
<tr>
<td>+/-0</td>
<td>-0</td>
<td>+/- pi</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>+/-inf</td>
<td>+0 / +pi</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>+/-inf</td>
<td>-0 / -pi</td>
</tr>
<tr>
<td>+/-inf</td>
<td>+inf</td>
<td>+/- (pi/4)</td>
</tr>
<tr>
<td>+/-inf</td>
<td>-inf</td>
<td>+/- (3*pi/4)</td>
</tr>
</tbody>
</table>

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.

References

[1]

Examples

Consider four points in different quadrants:

```python
>>> x = np.array([-1, +1, +1, -1])  # doctest: +SKIP
>>> y = np.array([-1, -1, +1, +1])  # doctest: +SKIP
>>> np.arctan2(y, x) * 180 / np.pi  # doctest: +SKIP
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]:

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.
dask.array.arctan2(x, y)  # doctest: +SKIP
array([-1.57079633,  1.57079633])

```
>>> np.arctan2([1., -1.], [0., 0.])  # doctest: +SKIP
array([ 1.57079633, -1.57079633])
```

Inverse hyperbolic tangent element-wise.

**Parameters**

- `x` [array_like] Input array.
- `y` [array_like] Other input array.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs** For other keyword-only arguments, see the ufunc docs.

**Returns**

- `out` [ndarray or scalar] Array of the same shape as `x`. This is a scalar if `x` is a scalar.

**See also:**

`emath.arctanh`

**Notes**

arctanh is a multivalued function: for each `x` there are infinitely many numbers `z` such that tanh(z) = x. The convention is to return the `z` whose imaginary part lies in [-pi/2, pi/2].

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

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

The inverse hyperbolic tangent is also known as atanh or tanh^-1.

**References**

[1], [2]

**Examples**

```
>>> np.arctanh([[0, -0.5], [0, -0.5]])  # doctest: +SKIP
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

```python
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])
```

Indexes of the maximal elements of a N-dimensional array:

```python
>>> ind = np.unravel_index(np.argmax(a, axis=None), a.shape)
>>> ind
(1, 2)
>>> a[ind]
15
```

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

dask.array.argmin (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**

```python
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmin(a)
0
>>> np.argmin(a, axis=0)
array([0, 0, 0])
>>> np.argmin(a, axis=1)
array([0, 0])
```

Indices of the minimum elements of a N-dimensional array:

```python
>>> ind = np.unravel_index(np.argmin(a, axis=None), a.shape)
>>> ind
(0, 0)
>>> a[ind]
10
```

```python
>>> b = np.arange(6) + 10
>>> b[4] = 10
>>> b
array([10, 11, 12, 13, 10, 15])
>>> np.argmin(b)  # Only the first occurrence is returned.
0
```

dask.array.argtopk(a, k, axis=-1, split_every=None)

Extract the indices of the k largest elements from a on the given axis, and return them sorted from largest to smallest. If k is negative, extract the indices of the -k smallest elements instead, and return them sorted from smallest to largest.
This performs best when \( k \) is much smaller than the chunk size. All results will be returned in a single chunk along the given axis.

### Parameters

- **x**: Array  Data being sorted
- **k**: int
- **axis**: int, optional
- **split_every**: int \( \geq 2 \), optional  See `topk()`. The performance considerations for `topk` also apply here.

### Returns

Selection of np.intp indices of `x` with size \( \text{abs}(k) \) along the given axis.

### Examples

```python
>>> import dask.array as da
>>> x = np.array([5, 1, 3, 6])
>>> d = da.from_array(x, chunks=2)
>>> d.argtopk(2).compute()
array([3, 0])
>>> d.argtopk(-2).compute()
array([1, 2])
```

dask.array.argwhere(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.argwhere(a) is the same as np.transpose(np.nonzero(a)).

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

### Examples

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

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

References

[1], [2]

Examples

```python
>>> np.around([0.37, 1.64])
array([ 0., 2.])
>>> np.around([0.37, 1.64], decimals=1)
array([ 0.4, 1.6])
>>> np.around([1.5, 2.5, 3.5, 4.5])  # rounds to nearest even value
array([ 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='K', 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* [{‘K’, ‘A’, ‘C’, ‘F’}, optional] Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless ‘F’ is specified, in which case it will be in Fortran order (column major). If object is an array the following holds.

<table>
<thead>
<tr>
<th>order</th>
<th>no copy</th>
<th>copy=True</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘K’</td>
<td>unchanged</td>
<td>F &amp; C order preserved, otherwise most similar order</td>
</tr>
<tr>
<td>‘A’</td>
<td>unchanged</td>
<td>F order if input is F and not C, otherwise C order</td>
</tr>
<tr>
<td>‘C’</td>
<td>C order</td>
<td>C order</td>
</tr>
<tr>
<td>‘F’</td>
<td>F order</td>
<td>F order</td>
</tr>
</tbody>
</table>

When copy=False and a copy is made for other reasons, the result is the same as if copy=True, with some exceptions for A, see the Notes section. The default order is ‘K’.

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

*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.

*full_like* Return a new array with shape of input filled with value.

*empty* Return a new uninitialized array.

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

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

*full* Return a new array of given shape filled with value.
Notes

When order is ‘A’ and object is an array in neither ‘C’ nor ‘F’ order, and a copy is forced by a change in dtype, then the order of the result is not necessarily ‘C’ as expected. This is likely a bug.

Examples

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

Upcasting:

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

More than one dimension:

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

Minimum dimensions 2:

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

Type provided:

```python
>>> 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:

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

Creating an array from sub-classes:

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

```python
>>> 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

Examples

```python
g>>> import dask.array as da
g>>> import numpy as np
g>>> x = np.arange(3)
g>>> da.asanyarray(x)
dask.array<array, shape=(3,), dtype=int64, chunksize=(3,)>
g
>>> y = [[1, 2, 3], [4, 5, 6]]
g>>> da.asanyarray(y)
dask.array<array, shape=(2, 3), dtype=int64, chunksize=(2, 3)>
```

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

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


Examples

```python
g>>> import dask.array as da
g>>> import numpy as np
g>>> x = np.arange(3)
g>>> da.asarray(x)
dask.array<array, shape=(3,), dtype=int64, chunksize=(3,)>
g
>>> y = [[1, 2, 3], [4, 5, 6]]
g>>> 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 list of arrays, each with a.ndim >= 1. Copies are made only if necessary.

See also:

atleast_2d, atleast_3d
Examples

```python
>>> 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 list 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

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

>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_2d(x)
array([[0., 1., 2.],
       [3., 4., 5.],
       [6., 7., 8.]])
>>> 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(*arys)
View inputs as arrays with at least three dimensions.

Parameters

arys1, arys2, ... [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 list 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**

```python
>>> 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.base  # x is a reshape, so not base itself
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.average(a, axis=None, weights=None, returned=False)

Compute the weighted average along the specified axis.

**Parameters**

- **a**  [array_like] Array containing data to be averaged. If `a` is not an array, a conversion is attempted.
- **axis**  [None or int or tuple of ints, optional] Axis or axes along which to average `a`. The default, `axis=None`, will average over 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, averaging is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

- **weights**  [array_like, optional] An array of weights associated with the values in `a`. Each value in `a` contributes to the average according to its associated weight. The weights array can either be 1-D (in which case its length must be the size of `a` along the given axis) or of the same shape as `a`. If `weights=None`, then all data in `a` are assumed to have a weight equal to one.
**returned** [bool, optional] Default is *False*. If *True*, the tuple \((\text{average}, \text{sum\_of\_weights})\) is returned, otherwise only the average is returned. If \(\text{weights}=None\), \(\text{sum\_of\_weights}\) is equivalent to the number of elements over which the average is taken.

**Returns**

\(\text{retval, [sum\_of\_weights]}\) [array\_type or double] Return the average along the specified axis. When \(\text{returned}\) is *True*, return a tuple with the average as the first element and the sum of the weights as the second element. \(\text{sum\_of\_weights}\) is of the same type as \(\text{retval}\). The result dtype follows a general pattern. If \(\text{weights}\) is None, the result dtype will be that of \(a\), or float64 if \(a\) is integral. Otherwise, if \(\text{weights}\) is not None and \(a\) is non-integral, the result type will be the type of lowest precision capable of representing values of both \(a\) and \(\text{weights}\). If \(a\) happens to be integral, the previous rules still applies but the result dtype will at least be float64.

**Raises**

- **ZeroDivisionError** When all weights along axis are zero. See *numpy.ma.average* for a version robust to this type of error.
- **TypeError** When the length of 1D \(\text{weights}\) is not the same as the shape of \(a\) along axis.

**See also:**

- *mean*
- *numpy\_result\_type* Returns the type that results from applying the numpy type promotion rules to the arguments.

**Examples**

```python
>>> data = range(1,5)
>>> np.average(data)
2.5
>>> np.average(range(1,11), weights=range(10,0,-1))
4.0
```

```python
>>> data = np.arange(6).reshape((3,2))
>>> np.average(data, axis=1, weights=[1./4, 3./4])
array([ 0.75, 2.75, 4.75])
>>> np.average(data, weights=[1./4, 3./4])
Traceback (most recent call last): 
  ... TypeError: Axis must be specified when shapes of a and weights differ.
```

```python
>>> a = np.ones(5, dtype=np.float128)
>>> w = np.ones(5, dtype=np.complex64)
>>> avg = np.average(a, weights=w)
>>> print(avg.dtype)
complex256
```
dask.array.bincount(x, weights=None, minlength=0)

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 negative.
- TypeError If the type of the input is float or complex.

See also:

- histogram, digitize, unique

Examples

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

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

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

```python
>>> np.bincount(np.arange(5, dtype=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.

```python
>>> 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)
aarray([ 0.3, 0.7, 1.1])
```
Dask array bitwise_and:
```
dask.array.bitwise_and(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj])
```

- **Compute the bit-wise AND of two arrays element-wise.**
- Computes the bit-wise AND of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `&`.

**Parameters**
- `x1, x2` [array_like] Only integer and boolean types are handled.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.**

**Returns**
- `out` [ndarray or scalar] Result. This is a scalar if both `x1` and `x2` are scalars.

**See also:**
- `logical_and`, `bitwise_or`, `bitwise_xor`
- `binary_repr` Return the binary representation of the input number as a string.

**Examples**

The number 13 is represented by `00001101`. Likewise, 17 is represented by `00010001`. The bit-wise AND of 13 and 17 is therefore `000000001`, or 1:

```python
>>> np.bitwise_and(13, 17)  # doctest: +SKIP
1
```

```python
>>> np.bitwise_and(14, 13)  # doctest: +SKIP
12
>>> np.bitwise_and([14,3], 13)  # doctest: +SKIP
array([12, 1])
```

```python
>>> np.bitwise_and([11,7], [4,25])  # doctest: +SKIP
array([0, 1])
>>> np.bitwise_and(np.array([2,5,255]), np.array([3,14,16]))  # doctest: +SKIP
array([2, 4, 16])
>>> np.bitwise_and([True, True], [False, True])  # doctest: +SKIP
array([False, True])
```

Dask array bitwise_not:
```
dask.array.bitwise_not(x, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj])
```

- **Compute bit-wise inversion, or bit-wise NOT, element-wise.**
- Computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `~`.  

---

4.7. Array 91
For signed integer inputs, the two’s complement is returned. In a two’s-complement system negative numbers are represented by the two’s complement of the absolute value. This is the most common method of representing signed integers on computers [1]. A N-bit two’s-complement system can represent every integer in the range $-2^{N-1}$ to $+2^{N-1} - 1$.

Parameters

x [array_like] Only integer and boolean types are handled.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Result. This is a scalar if x is a scalar.

See also:

bitwise_and, bitwise_or, bitwise_xor, logical_not

binary_repr Return the binary representation of the input number as a string.

Notes

bitwise_not is an alias for invert:

```python
>>> np.bitwise_not is np.invert  # doctest: +SKIP
True
```

References

[1]

Examples

We’ve seen that 13 is represented by 00001101. The invert or bit-wise NOT of 13 is then:

```python
>>> np.invert(np.array([13], dtype=uint8))  # doctest: +SKIP
array([242], dtype=uint8)
>>> np.binary_repr(x, width=8)  # doctest: +SKIP
'00001101'
>>> np.binary_repr(242, width=8)  # doctest: +SKIP
'11110010'
```

The result depends on the bit-width:

```python
>>> np.invert(np.array([13], dtype=uint16))  # doctest: +SKIP
array([65522], dtype=uint16)
>>> np.binary_repr(x, width=16)  # doctest: +SKIP
'0000000000001101'
```
>>> np.binary_repr(65522, width=16)  # doctest: +SKIP
'1111111111110010'

When using signed integer types the result is the two’s complement of the result for the unsigned type:

```python
>>> np.invert(np.array([13], dtype=int8))  # doctest: +SKIP
array([-14], dtype=int8)
>>> np.binary_repr(-14, width=8)  # doctest: +SKIP
'11110010'
```

Booleans are accepted as well:

```python
>>> np.invert(array([True, False]))  # doctest: +SKIP
array([False, True])
```

```python
dask.array.bitwise_or(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])
```

Compute the bit-wise OR of two arrays element-wise.

Computes the bit-wise OR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `|`.

**Parameters**

- `x1, x2` [array_like] Only integer and boolean types are handled.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

**Returns**

- `out` [ndarray or scalar] Result. This is a scalar if both `x1` and `x2` are scalars.

**See also:**

- `logical_or`, `bitwise_and`, `bitwise_xor`
- `binary_repr` Return the binary representation of the input number as a string.

**Examples**

The number 13 has the binary representation `00001101`. Likewise, 16 is represented by `00010000`. The bit-wise OR of 13 and 16 is then `00011101`, or 29:

```python
>>> np.bitwise_or(13, 16)  # doctest: +SKIP
29
>>> np.binary_repr(29)  # doctest: +SKIP
'11101'
```
dask.array.bitwise_xor(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature, extobj)

Compute the bit-wise XOR of two arrays element-wise.

Computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ^.

Parameters

- **x1, x2** [array_like] Only integer and boolean types are handled.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **out** [ndarray or scalar] Result. This is a scalar if both x1 and x2 are scalars.

See also:

- **logical_xor**, **bitwise_and**, **bitwise_or**
- **binary_repr** Return the binary representation of the input number as a string.

Examples

The number 13 is represented by 00001101. Likewise, 17 is represented by 00010001. The bit-wise XOR of 13 and 17 is therefore 00011100, or 28:
```python
>>> np.bitwise_xor(31, 5)  # doctest: +SKIP
26
>>> np.bitwise_xor([31,3], 5)  # doctest: +SKIP
array([26, 6])
```

dask.array.block (arrays, allow_unknown_chunksizes=False)

Assemble an nd-array from nested lists of blocks.

Blocks in the innermost lists are concatenated along the last dimension (-1), then these are concatenated 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.

4.7. Array 95
Notes

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

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

\begin{verbatim}
AAAAbb
AAAAbb
cccDDDD
\end{verbatim}

But is also allowed to produce, for some \(a, b, c, d\):

\begin{verbatim}
AAAAbb
AAAAbb
cDDDDD
\end{verbatim}

Since concatenation happens along the last axis first, \texttt{block} is \texttt{not} capable of producing the following directly:

\begin{verbatim}
AAAAbb
cccbbb
cccDD
\end{verbatim}

Matlab’s “square bracket stacking”, \([A, B, \ldots; p, q, \ldots]\), is equivalent to \texttt{block}([[\(A, B, \ldots\)], [\(p, q, \ldots\)]]).

\texttt{dask.array.broadcast_arrays (**args, **kwargs)}

Broadcast any number of arrays against each other.

Parameters

- \texttt{**args} [array_likes] The arrays to broadcast.
- \texttt{subok} [bool, optional] If True, then sub-classes will be passed-through, otherwise the returned arrays will be forced to be a base-class array (default).

Returns

- \texttt{broadcasted} [list of arrays] These arrays are views on the original arrays. They are typically not contiguous. Furthermore, more than one element of a broadcasted array may refer to a single memory location. If you need to write to the arrays, make copies first.

Examples

```python
>>> x = np.array([[1, 2, 3]])
>>> y = np.array([[4], [5]])
>>> np.broadcast_arrays(x, y)
[array([[1, 2, 3],
        [1, 2, 3]]), array([[4, 4, 4],
        [5, 5, 5]])]
```

Here is a useful idiom for getting contiguous copies instead of non-contiguous views.

```python
>>> [np.array(a) for a in np.broadcast_arrays(x, y)]
[array([[1, 2, 3],
        [1, 2, 3]]), array([[4, 4, 4],
        [5, 5, 5]])]
```
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()

4.7. Array 97
dask.array.ceil(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature, extobj)

Return the ceiling of the input, element-wise.

The ceiling of the scalar \( x \) is the smallest integer \( i \), such that \( i \geq x \). It is often denoted as \( \lceil x \rceil \).

**Parameters**

- **x** [array_like] Input data.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

**Returns**

- **y** [ndarray or scalar] The ceiling of each element in \( x \), with float dtype. This is a scalar if \( x \) is a scalar.

See also:

floor, trunc, rint

**Examples**

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])  # doctest: +SKIP
>>> np.ceil(a)  # doctest: +SKIP
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 \( Bchoices[i] \), \( i = 0, \ldots, n-1 \) we have that, necessarily, \( Ba.shape == Bchoices[i].shape \) for each \( i \). Then, a new array with shape \( Ba.shape \) is created as follows:

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

**Parameters**
a [int array] This array must contain integers in \([0, n-1]\), where \(n\) is the number of choices, unless \texttt{mode=wrap} or \texttt{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 \texttt{choices} is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to \texttt{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

\texttt{ValueError: shape mismatch} If \(a\) and each choice array are not all broadcastable to the same shape.

See also:

\texttt{ndarray.choose} equivalent method

Notes

To reduce the chance of misinterpretation, even though the following “abuse” is nominally supported, \texttt{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

```python
>>> 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:
```python
>>> 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]])
```

```python
>>> # 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 or None] Minimum value. If None, clipping is not performed on lower interval edge. Not more than one of a_min and a_max may be None.
- **a_max** [scalar or array_like or None] Maximum value. If None, clipping is not performed on upper interval edge. Not more than one of a_min and a_max may be None. If a_min or a_max are array_like, then the three arrays will be broadcasted to match their shapes.
- **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**

```python
>>> a = np.arange(10)  # doctest: +SKIP
>>> np.clip(a, 1, 8)  # doctest: +SKIP
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
```

(continues on next page)
>>> a  # doctest: +SKIP
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3, 4, 1, 1, 4, 4, 4, 4, 4, 8], 8)  # doctest: +SKIP
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

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

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

```python
>>> 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

```python
x.conj(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])
```

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.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The complex conjugate of x, with same dtype as y. This is a scalar if x is a scalar.

Examples

```python
>>> np.conjugate(1+2j)  # doctest: +SKIP
(1-2j)
```

```python
dx = np.eye(2) + 1j * np.eye(2)  # doctest: +SKIP
deprecated
```  

dask.array.copysign(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] The values of x1 with the sign of x2. This is a scalar if both x1 and x2 are scalars.

Examples

```python
>>> np.copysign(1.3, -1)  # doctest: +SKIP
-1.3
```

```python
>>> 1.0/np.copysign(0, 1)  # doctest: +SKIP
inf
```

```python
>>> 1.0/np.copysign(0, -1)  # doctest: +SKIP
-inf
```

```python
>>> np.copysign([[-1, 0, 1], [-1.1]], -1.1)  # doctest: +SKIP
array([-1, 0, 1])
```

```python
>>> np.copysign([-1, 0, 1], np.arange(3)-1)  # doctest: +SKIP
array([-1, 0, 1])
```
dask.array.corrcoef(x, y=None, rowvar=True, bias=<no value>, ddof=<no value>)

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

\[ R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} \cdot C_{jj}}} \]

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** [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** ["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 abs(a) <= 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=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Cosine element-wise.

**Parameters**

- **x** [array_like] Input array in radians.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] The corresponding cosine values. This is a scalar if x is a scalar.

Notes

If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)

References


Examples

```python
>>> np.cos(np.array([0, np.pi/2, np.pi])) # doctest: +SKIP
array([ 1.00000000e+00, 6.12303177e-17, -1.00000000e+00])
```

Hyperbolic cosine, element-wise.

Equivalent to \( \frac{1}{2} (\exp(x) + \exp(-x)) \) and \( \cos(1j*x) \).

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Output array of same shape as x. This is a scalar if x is a scalar.
Examples

```python
>>> np.cosh(0)  # doctest: +SKIP
1.0
```

The hyperbolic cosine describes the shape of a hanging cable:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> x = np.linspace(-4, 4, 1000)  # doctest: +SKIP
>>> plt.plot(x, np.cosh(x))  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

dask.array.count_nonzero(a, axis=None)

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

The word “non-zero” is in reference to the Python 2.x built-in method `__nonzero__()` (renamed
`__bool__()` in Python 3.x) of Python objects that tests an object’s “truthfulness”. For example, any num-
ber is considered truthful if it is nonzero, whereas any string is considered truthful if it is not the empty
string. Thus, this function (recursively) counts how many elements in `a` (and in sub-arrays thereof) have their
`__nonzero__()` or `__bool__()` method evaluated to `True`.

**Parameters**

- **a** [array_like] The array for which to count non-zeros.
- **axis** [int or tuple, optional] Axis or tuple of axes along which to count non-zeros. Default is
  None, meaning that non-zeros will be counted along a flattened version of `a`.

  New in version 1.12.0.

**Returns**

**count** [int or array of int] Number of non-zero values in the array along a given axis. Otherwise,
the total number of non-zero values in the array is returned.

**See also:**

- **nonzero** Return the coordinates of all the non-zero values.

**Examples**

```python
>>> np.count_nonzero(np.eye(4))
4
```

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

```python
>>> np.count_nonzero([[0,1,7,0,0],[3,0,0,2,19]], axis=0)
array([1, 1, 1, 1, 1])
```

```python
>>> np.count_nonzero([[0,1,7,0,0],[3,0,0,2,19]], axis=1)
array([5, 3])
```

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,
\( X = [x_1, x_2, \ldots, x_N]^T \), 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 repre-
  sents 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 >= 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 con-
  sidered 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> X = np.stack((x, y), axis=0)
>>> 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

```python
>>> 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`:

```python
>>> 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`:

```python
>>> 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

- `csum_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

```python
>>> 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]])
```

```
import dask.array as da

dask.array.deg2rad(x[, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True][, signature, extobj])
```

Convert angles from degrees to radians.

**Parameters**

- `x` [array_like] Angles in degrees.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

**Returns**

- `y` [ndarray] The corresponding angle in radians. This is a scalar if `x` is a scalar.

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

```python
>>> np.deg2rad(180)  # doctest: +SKIP
3.1415926535897931
```

dask.array.degrees(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

Convert angles from radians to degrees.

Parameters

x [array_like] Input array in radians.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray of floats] The corresponding degree values; if out was supplied this is a reference to it. This is a scalar if x is a scalar.

See also:

rad2deg equivalent function

Examples

Convert a radian array to degrees

```python
>>> rad = np.arange(12.)*np.pi/6  # doctest: +SKIP
>>> np.degrees(rad)  # doctest: +SKIP
array([ 0., 30., 60., 90., 120., 150., 180., 210., 240.,
       270., 300., 330.])
```

```python
>>> out = np.zeros((rad.shape))  # doctest: +SKIP
>>> r = degrees(rad, out)  # doctest: +SKIP
>>> np.all(r == out)  # doctest: +SKIP
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

```python
>>> 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.diagonal (a, offset=0, axis1=0, axis2=1)
Return specified diagonals.

If a is 2-D, returns the diagonal of a with the given offset, i.e., the collection of elements of the form a[i, i+offset]. If a has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing axis1 and axis2 and appending an index to the right equal to the size of the resulting diagonals.

In versions of NumPy prior to 1.7, this function always returned a new, independent array containing a copy of the values in the diagonal.

In NumPy 1.7 and 1.8, it continues to return a copy of the diagonal, but depending on this fact is deprecated. Writing to the resulting array continues to work as it used to, but a FutureWarning is issued.

Starting in NumPy 1.9 it returns a read-only view on the original array. Attempting to write to the resulting array will produce an error.

In some future release, it will return a read/write view and writing to the returned array will alter your original array. The returned array will have the same type as the input array.

If you don’t write to the array returned by this function, then you can just ignore all of the above.
If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e., use `np.diagonal(a).copy()` instead of just `np.diagonal(a)`. This will work with both past and future versions of NumPy.

**Parameters**

- **a** [array_like] Array from which the diagonals are taken.
- **offset** [int, optional] Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).
- **axis1** [int, optional] Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).
- **axis2** [int, optional] Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis (1).

**Returns**

- **array_of_diagonals** [ndarray] If `a` is 2-D, then a 1-D array containing the diagonal and of the same type as `a` is returned unless `a` is a `matrix`, in which case a 1-D array rather than a (2-D) `matrix` is returned in order to maintain backward compatibility.

If `a.ndim > 2`, then the dimensions specified by `axis1` and `axis2` are removed, and a new axis inserted at the end corresponding to the diagonal.

**Raises**

- **ValueError** If the dimension of `a` is less than 2.

**See also:**

- `diag` MATLAB work-a-like for 1-D and 2-D arrays.
- `diagflat` Create diagonal arrays.
- `trace` Sum along diagonals.

**Examples**

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

A 3-D example:

```python
>>> a = np.arange(8).reshape(2,2,2); a
array([[[0, 1],
       [2, 3]],
       [[4, 5],
       [6, 7]]])
>>> a.diagonal(0, # Main diagonals of two arrays created by skipping
... 0, # across the outer(left)-most axis last and
... 1) # the "middle" (row) axis first.
array([[0, 6],
       [1, 7]])
```
The sub-arrays whose main diagonals we just obtained; note that each corresponds to fixing the right-most (column) axis, and that the diagonals are “packed” in rows.

```python
>>> a[:,:,0]  # main diagonal is [0 6]
array([[0, 2],
       [4, 6]])
>>> a[:,:,1]  # main diagonal is [1 7]
array([[1, 3],
       [5, 7]])
```

dask.array.diff(a, n=1, axis=-1, prepend=<no value>, append=<no value>)

Calculate the n-th discrete difference along the 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. If zero, the input is returned as-is.
- **axis** [int, optional] The axis along which the difference is taken, default is the last axis.
- **prepend** [array_like, optional] Values to prepend or append to “a” along axis prior to performing the difference. Scalar values are expanded to arrays with length 1 in the direction of axis and the shape of the input array in along all other axes. Otherwise the dimension and shape must match “a” except along 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 \( n \). The type of the output is the same as the type of the difference between any two elements of \( a \). This is the same as the type of \( a \) in most cases. A notable exception is datetime64, which results in a timedelta64 output array.

**See also:**

gradient, ediff1d, cumsum

**Notes**

Type is preserved for boolean arrays, so the result will contain \( False \) when consecutive elements are the same and \( True \) when they differ.

For unsigned integer arrays, the results will also be unsigned. This should not be surprising, as the result is consistent with calculating the difference directly:

```python
>>> u8_arr = np.array([1, 0], dtype=np.uint8)
>>> np.diff(u8_arr)
array([255], dtype=uint8)
```

If this is not desirable, then the array should be cast to a larger integer type first:

```python
>>> i16_arr = u8_arr.astype(np.int16)
>>> np.diff(i16_arr)
array([-1], dtype=int16)
```
Examples

```python
>>> 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]])

>>> x = np.arange('1066-10-13', '1066-10-16', dtype=np.datetime64)
>>> np.diff(x)
array([1, 1], dtype='timedelta64[D]')
```

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

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

<table>
<thead>
<tr>
<th>right</th>
<th>order of bins</th>
<th>returned index $i$ satisfies</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>increasing</td>
<td>$\text{bins}[i-1] \leq x &lt; \text{bins}[i]$</td>
</tr>
<tr>
<td>True</td>
<td>increasing</td>
<td>$\text{bins}[i-1] &lt; x \leq \text{bins}[i]$</td>
</tr>
<tr>
<td>False</td>
<td>decreasing</td>
<td>$\text{bins}[i-1] &gt; x \geq \text{bins}[i]$</td>
</tr>
<tr>
<td>True</td>
<td>decreasing</td>
<td>$\text{bins}[i-1] \geq x &gt; \text{bins}[i]$</td>
</tr>
</tbody>
</table>

If values in $x$ are beyond the bounds of $bins$, 0 or len(bins) is returned as appropriate.

**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**

- **indices** [ndarray of ints] Output array of indices, of same shape as $x$.

**Raises**

- **ValueError** If $bins$ is not monotonic.
- **TypeError** If the type of the input is complex.

**See also:**

bincount, histogram, unique, searchsorted
Notes

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

New in version 1.10.0.

\( \text{np.digitize} \) is implemented in terms of \( \text{np.searchsorted} \). 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.

For monotonically _increasing_ \( bins \), the following are equivalent:

```python
np.digitize(x, bins, right=True)
np.searchsorted(bins, x, side='left')
```

Note that as the order of the arguments are reversed, the side must be too. The \( \text{searchsorted} \) call is marginally faster, as it does not do any monotonicity checks. Perhaps more importantly, it supports all dtypes.

Examples

```python
>>> 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
```

```python
>>> 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. Specifically,

- If both \( a \) and \( b \) are 1-D arrays, it is inner product of vectors (without complex conjugation).
  - If both \( a \) and \( b \) are 2-D arrays, it is matrix multiplication, but using \( \text{matmul()} \) or \( a \ @ \ b \) is preferred.
  - If either \( a \) or \( b \) is 0-D (scalar), it is equivalent to \( \text{multiply()} \) and using \( \text{numpy.multiply(a, b)} \) or \( a \ * \ b \) is preferred.
  - If \( a \) is an N-D array and \( b \) is a 1-D array, it is a sum product over the last axis of \( a \) and \( b \).
  - If \( a \) is an N-D array and \( b \) is an M-D array (where \( M \geq 2 \)), it is a sum product over the last axis of \( a \) and the second-to-last axis of \( b \):

```python
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 \( \text{dot}(a,b) \). 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 \( \text{out} \) 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

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

Neither argument is complex-conjugated:

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

For 2-D arrays it is the matrix product:

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

```python
>>> 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,:,:]
499128
```

```python
>>> sum(a[2,3,2,:] * b[1,2,:,:])
499128
```

dask.array.dstack(tup)

Stack arrays in sequence depth wise (along third axis).
This is equivalent to concatenation along the third axis after 2-D arrays of shape \((M,N)\) have been reshaped to \((M,N,1)\) and 1-D arrays of shape \((N,)\) have been reshaped to \((1,N,1)\). Rebuilds arrays divided by \texttt{dsplit}.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions \textit{concatenate}, \textit{stack} and \textit{block} provide more general stacking and concatenation operations.

**Parameters**

- \texttt{tup} [sequence of arrays] The arrays must have the same shape along all but the third axis. 1-D or 2-D arrays must have the same shape.

**Returns**

- \texttt{stacked} [ndarray] The array formed by stacking the given arrays, will be at least 3-D.

**See also:**

- \texttt{stack} Join a sequence of arrays along a new axis.
- \texttt{vstack} Stack along first axis.
- \texttt{hstack} Stack along second axis.
- \texttt{concatenate} Join a sequence of arrays along an existing axis.
- \texttt{dsplit} Split array along third axis.

**Examples**

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

```python
>>> 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.\texttt{ediff1d}(ary, to_end=None, to_begin=None)

The differences between consecutive elements of an array.

**Parameters**

- \texttt{ary} [array_like] If necessary, will be flattened before the differences are taken.

- \texttt{to_end} [array_like, optional] Number(s) to append at the end of the returned differences.

- \texttt{to_begin} [array_like, optional] Number(s) to prepend at the beginning of the returned differences.

**Returns**

- \texttt{ediff1d} [ndarray] The differences. Loosely, this is \texttt{ary.flat[1:] - ary.flat[:-1]}.

**See also:**

- \texttt{diff}, \texttt{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

```python
>>> 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.

```python
>>> 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, e.g., (2, 3) or 2.
- **dtype** [data-type, optional] Desired output data-type for the array, e.g, `numpy.int8`. Default is `numpy.float64`.
- **order** [{‘C’, ‘F’}, optional, default: ‘C’] 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** Return an empty array with shape and type of input.
- **ones** Return a new array setting values to one.
- **zeros** Return a new array setting values to zero.
- **full** Return a new array of given shape filled with value.

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

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

```python
>>> 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.einsum(subscripts, *operands, out=None, dtype=None, order='K', casting='safe', optimize=False)

Evaluates the Einstein summation convention on the operands.

Using the Einstein summation convention, many common multi-dimensional, linear algebraic array operations can be represented in a simple fashion. In *implicit* mode `einsum` computes these values.

In *explicit* mode, `einsum` provides further flexibility to compute other array operations that might not be considered classical Einstein summation operations, by disabling, or forcing summation over specified subscript labels.

See the notes and examples for clarification.

**Parameters**

- `subscripts` [str] Specifies the subscripts for summation as comma separated list of subscript labels. An implicit (classical Einstein summation) calculation is performed unless the explicit indicator `->` is included as well as subscript labels of the precise output form.
operands [list of array_like] These are the arrays for the operation.

out [ndarray, optional] If provided, the calculation is done into this array.

dtype [[data-type, None], optional] If provided, forces the calculation to use the data type specified. Note that you may have to also give a more liberal casting parameter to allow the conversions. Default is None.

order [{‘C’, ‘F’, ‘A’, ‘K’}, optional] Controls the memory layout of the output. ‘C’ means it should be C contiguous. ‘F’ means it should be Fortran contiguous, ‘A’ means it should be ‘F’ if the inputs are all ‘F’, ‘C’ otherwise, ‘K’ means it should be as close to the layout as the inputs as is possible, including arbitrarily permuted axes. Default is ‘K’.

casting [{‘no’, ‘equiv’, ‘safe’, ‘same_kind’, ‘unsafe’}, optional] Controls what kind of data casting may occur. Setting this to ‘unsafe’ is not recommended, as it can adversely affect accumulations.

- ‘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.

Default is ‘safe’.

optimize [{False, True, ‘greedy’, ‘optimal’}, optional] Controls if intermediate optimization should occur. No optimization will occur if False and True will default to the ‘greedy’ algorithm. Also accepts an explicit contraction list from the np.einsum_path function. See np.einsum_path for more details. Defaults to False.

Returns


See also:

einsum_path, dot, inner, outer, tensordot, linalg.multi_dot

Notes

New in version 1.6.0.

The Einstein summation convention can be used to compute many multi-dimensional, linear algebraic array operations. einsum provides a succinct way of representing these.

A non-exhaustive list of these operations, which can be computed by einsum, is shown below along with examples:

- Trace of an array, numpy.trace().
- Return a diagonal, numpy.diag().
- Array axis summations, numpy.sum().
- Transpositions and permutations, numpy.transpose().
- Matrix multiplication and dot product, numpy.matmul() numpy.dot().
- Vector inner and outer products, numpy.inner() numpy.outer().
• Broadcasting, element-wise and scalar multiplication, `numpy.multiply()`.
• Tensor contractions, `numpy.tensordot()`.
• Chained array operations, in efficient calculation order, `numpy.einsum_path()`.

The subscripts string is a comma-separated list of subscript labels, where each label refers to a dimension of the corresponding operand. Whenever a label is repeated it is summed, so `np.einsum('i,i', a, b)` is equivalent to `np.inner(a, b)`. If a label appears only once, it is not summed, so `np.einsum('i', a)` produces a view of `a` with no changes. A further example `np.einsum('ij,jk', a, b)` describes traditional matrix multiplication and is equivalent to `np.matmul(a, b)`. Repeated subscripts in one operand take the diagonal. For example, `np.einsum('ii', a)` is equivalent to `np.trace(a)`.

In *implicit mode*, the chosen subscripts are important since the axes of the output are reordered alphabetically. This means that `np.einsum('ij', a)` doesn’t affect a 2D array, while `np.einsum('ji', a)` takes its transpose. Additionally, `np.einsum('ij,jk', a, b)` returns a matrix multiplication, while, `np.einsum('ij,jh', a, b)` returns the transpose of the multiplication since subscript ‘`h`’ precedes subscript ‘`i`’.

In *explicit mode* the output can be directly controlled by specifying output subscript labels. This requires the identifier ‘`->`’ as well as the list of output subscript labels. This feature increases the flexibility of the function since summing can be disabled or forced when required. The call `np.einsum('i->', a)` is like `np.sum(a, axis=-1)`, and `np.einsum('ii->i', a)` is like `np.diag(a)`. The difference is that `einsum` does not allow broadcasting by default. Additionally `np.einsum('ij,jh->ih', a, b)` directly specifies the order of the output subscript labels and therefore returns matrix multiplication, unlike the example above in implicit mode.

To enable and control broadcasting, use an ellipsis. Default NumPy-style broadcasting is done by adding an ellipsis to the left of each term, like `np.einsum('...ii->...i', a)`. To take the trace along the first and last axes, you can do `np.einsum('i...i', a)`, or to do a matrix-matrix product with the left-most indices instead of rightmost, one can do `np.einsum('ij...,jk...->ik...', a, b)`.

When there is only one operand, no axes are summed, and no output parameter is provided, a view into the operand is returned instead of a new array. Thus, taking the diagonal as `np.einsum('ii->i', a)` produces a view (changed in version 1.10.0).

`einsum` also provides an alternative way to provide the subscripts and operands as `einsum(op0, sublist0, op1, sublist1, ..., [sublistout])`. If the output shape is not provided in this format `einsum` will be calculated in implicit mode, otherwise it will be performed explicitly. The examples below have corresponding `einsum` calls with the two parameter methods.

New in version 1.10.0.

Views returned from einsum are now writeable whenever the input array is writeable. For example, `np.einsum('ijk...->kji...', a)` will now have the same effect as `np.swapaxes(a, 0, 2)` and `np.einsum('ii->i', a)` will return a writeable view of the diagonal of a 2D array.

New in version 1.12.0.

Added the `optimize` argument which will optimize the contraction order of an einsum expression. For a contraction with three or more operands this can greatly increase the computational efficiency at the cost of a larger memory footprint during computation.

Typically a ‘greedy’ algorithm is applied which empirical tests have shown returns the optimal path in the majority of cases. In some cases ‘optimal’ will return the superlative path through a more expensive, exhaustive search. For iterative calculations it may be advisable to calculate the optimal path once and reuse that path by supplying it as an argument. An example is given below.

See `numpy.einsum_path()` for more details.
Examples

```python
>>> a = np.arange(25).reshape(5,5)
>>> b = np.arange(5)
>>> c = np.arange(6).reshape(2,3)

Trace of a matrix:

```python
>>> np.einsum('ii', a)
60
>>> np.einsum(a, [0,0])
60
>>> np.trace(a)
60
```

Extract the diagonal (requires explicit form):

```python
>>> np.einsum('ii->i', a)
array([ 0, 6, 12, 18, 24])
>>> np.einsum(a, [0,0], [0])
array([ 0, 6, 12, 18, 24])
>>> np.diag(a)
array([ 0, 6, 12, 18, 24])
```

Sum over an axis (requires explicit form):

```python
>>> np.einsum('ij->i', a)
array([ 10, 35, 60, 85, 110])
>>> np.einsum(a, [0,1], [0])
array([ 10, 35, 60, 85, 110])
>>> np.sum(a, axis=1)
array([ 10, 35, 60, 85, 110])
```

For higher dimensional arrays summing a single axis can be done with ellipsis:

```python
>>> np.einsum('...j->...', a)
array([ 10, 35, 60, 85, 110])
>>> np.einsum(a, [Ellipsis,1], [Ellipsis])
array([ 10, 35, 60, 85, 110])
```

Compute a matrix transpose, or reorder any number of axes:

```python
>>> np.einsum('ji', c)
array([[ 0, 3],
       [ 1, 4],
       [ 2, 5]])
>>> np.einsum('ij->ji', c)
array([[ 0, 3],
       [ 1, 4],
       [ 2, 5]])
>>> np.einsum(c, [1,0])
array([[ 0, 3],
       [ 1, 4],
       [ 2, 5]])
>>> np.transpose(c)
array([[ 0, 3],
       [ 1, 4],
       [ 2, 5]])
```
Vector inner products:

```python
gnp.einsum('i,i', b, b)
30
gnp.einsum(b, [0], b, [0])
30
np.inner(b, b)
30
```

Matrix vector multiplication:

```python
gnp.einsum('ij,j', a, b)
array([ 30,  80, 130, 180, 230])
gnp.einsum(a, [0, 1], b, [1])
array([ 30,  80, 130, 180, 230])
gnp.dot(a, b)
array([ 30,  80, 130, 180, 230])
gnp.einsum('...j,j', a, b)
array([ 30,  80, 130, 180, 230])
```

Broadcasting and scalar multiplication:

```python
gnp.einsum('..., ...', 3, c)
array([[ 0,  3,  6],
       [ 9, 12, 15]])
gnp.einsum(',ij', 3, c)
array([[ 0,  3,  6],
       [ 9, 12, 15]])
gnp.einsum(3, [Ellipsis], c, [Ellipsis])
array([[ 0,  3,  6],
       [ 9, 12, 15]])
gnp.multiply(3, c)
array([[ 0,  3,  6],
       [ 9, 12, 15]])
```

Vector outer product:

```python
gnp.einsum('i,j', np.arange(2)+1, b)
arra(y([[ 0,  1,  2,  3,  4],
       [ 8, 10, 12, 14, 16]])
gnp.einsum(np.arange(2)+1, [0], b, [1])
array([[ 0,  1,  2,  3,  4],
       [ 8, 10, 12, 14, 16]])
gnp.outer(np.arange(2)+1, b)
arra(y([[ 0,  1,  2,  3,  4],
       [ 8, 10, 12, 14, 16]])
```

Tensor contraction:

```python
gnp.einsum('ijk,jil->kl', a, b)
arra(y([[ 4400.,  4730.],
       [ 4532.,  4874.],
       [ 4664.,  5018.],
       [ 4796.,  5162.],
       [ 4928.,  5306.]])
```
array([[ 4400., 4730.],
       [ 4532., 4874.],
       [ 4664., 5018.],
       [ 4796., 5162.],
       [ 4928., 5306.]]))

>>> np.tensordot(a,b, axes=[[1,0],[0,1]])
array([[ 4400., 4730.],
       [ 4532., 4874.],
       [ 4664., 5018.],
       [ 4796., 5162.],
       [ 4928., 5306.]]))

Writeable returned arrays (since version 1.10.0):

>>> a = np.zeros((3, 3))
>>> np.einsum('ii->i', a)[:] = 1
>>> a
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]]))

Example of ellipsis use:

>>> a = np.arange(6).reshape((3,2))
>>> b = np.arange(12).reshape((4,3))
>>> np.einsum('ki,jk->ij', a, b)
array([[10, 28, 46, 64],
       [13, 40, 67, 94]])

Chained array operations. For more complicated contractions, speed ups might be achieved by repeatedly computing a ‘greedy’ path or pre-computing the ‘optimal’ path and repeatedly applying it, using an einsum_path insertion (since version 1.12.0). Performance improvements can be particularly significant with larger arrays:

```python
>>> a = np.ones(64).reshape(2,4,8)
# Basic `einsum`: ~1520ms (benchmarked on 3.1GHz Intel i5.)
>>> for iteration in range(500):
...   np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a)
# Sub-optimal `einsum` (due to repeated path calculation time): ~330ms
>>> for iteration in range(500):
...   np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize='optimal')
# Greedy `einsum` (faster optimal path approximation): ~160ms
>>> for iteration in range(500):
...   np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize='greedy')
# Optimal `einsum` (best usage pattern in some use cases): ~110ms
>>> path = np.einsum_path('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize=path)
>>> for iteration in range(500):
...   np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize=path)
```

dask.array.<code>exp</code>(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])
Calculate the exponential of all elements in the input array.

**Parameters**

- `x` [array_like] Input values.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- `**kwargs` For other keyword-only arguments, see the ufunc docs.

**Returns**

- `out` [ndarray or scalar] Output array, element-wise exponential of `x`. This is a scalar if `x` is a scalar.

**See also:**

- `expm1` Calculate $\exp(x) - 1$ for all elements in the array.
- `exp2` Calculate $2^{x}$ for all elements in the array.

**Notes**

The irrational number $e$ is also known as Euler’s number. It is approximately 2.718281, and is the base of the natural logarithm, $\ln$ (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, $\exp(x)$ is always positive.

For complex arguments, $x = a + ib$, we can write $e^x = e^a e^{ib}$. The first term, $e^a$, is already known (it is the real argument, described above). The second term, $e^{ib}$, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

**References**

[1], [2]

**Examples**

Plot the magnitude and phase of $\exp(x)$ in the complex plane:

```python
>>> import matplotlib.pyplot as plt
# doctest: +SKIP

>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)  # doctest: +SKIP
>>> xx = x + 1j * x[:, np.newaxis]  # a + ib over complex plane  # doctest: +SKIP
>>> out = np.exp(xx)  # doctest: +SKIP

>>> plt.subplot(121)  # doctest: +SKIP
>>> plt.imshow(np.abs(out),  # doctest: +SKIP...
... extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='gray')  # doctest: +SKIP
>>> plt.title('Magnitude of exp(x)')  # doctest: +SKIP
```
```python
>>> plt.subplot(122)  # doctest: +SKIP
>>> plt.imshow(np.angle(out),  # doctest: +SKIP
...     extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='hsv')
>>> plt.title('Phase (angle) of \exp(x)')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

dask.array.expm1(x, /, out=None, *, where=True, casting='same_kind', order='K', subok=True, signature=None, extobj=None)

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

Parameters

- **x** [array_like] Input values.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **out** [ndarray or scalar] Element-wise exponential minus one: out = \exp(x) - 1. This is a scalar if x is a scalar.

See also:

- **log1p** log(1 + x), the inverse of expm1.

Notes

This function provides greater precision than \exp(x) - 1 for small values of x.

Examples

The true value of \exp(1e-10) - 1 is 1.00000000005e-10 to about 32 significant digits. This example shows the superiority of expm1 in this case.

```python
>>> np.expm1(1e-10)  # doctest: +SKIP
1.00000000005e-10
>>> np.exp(1e-10) - 1  # doctest: +SKIP
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.

4.7. Array
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=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

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, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The absolute values of x, the returned values are always floats. This is a scalar if x is a scalar.

See also:

absolute Absolute values including complex types.

Examples

```python
>>> np.fabs(-1)  # doctest: +SKIP
1.0
>>> np.fabs([-1.2, 1.2])  # doctest: +SKIP
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**

```python
>>> np.fix(3.14)   # doctest: +SKIP
3.0
>>> np.fix(3)     # doctest: +SKIP
3.0
>>> np.fix([2.1, 2.9, -2.1, -2.9]) # doctest: +SKIP
array([ 2., 2., -2., -2.])
```

**dask.array.flatnonzero**

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

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

**Parameters**

- `a` [array_like] Input data.

**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**

```python
>>> 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:

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

**dask.array.flip**

Reverse element order along axis.

**Parameters**

- `axis` [int] Axis to reverse element order of.

**Returns**

- `reversed array` [ndarray]

**dask.array.flipud**

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:

flipud Flip array in the left/right direction.

rot90 Rotate array counterclockwise.

Notes

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

Examples

```python
>>> 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.]])
```

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

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

dask.array.flipud(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 m[:,::-1]. Requires the array to be at least 2-D.

Examples

```python
def fliplr(A):
    return A[:,::-1]
```

```
>>> 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.]])
```

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

dask.array.floor(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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.
- out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- y [ndarray or scalar] The floor of each element in x. This is a scalar if x is a scalar.

See also:

ceil, trunc, rint

Notes

Some spreadsheet programs calculate the “floor-towards-zero”, in other words $\text{floor}(-2.5) = -2$. NumPy instead uses the definition of floor where $\text{floor}(-2.5) = -3$.

Examples
```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])  # doctest: +SKIP
>>> np.floor(a)  # doctest: +SKIP
array([-2., -2., -1., 0., 1., 1., 2.])
```

dask.array.fmax(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

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.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [ndarray or scalar] The maximum of x1 and x2, element-wise. This is a scalar if both x1 and x2 are scalars.

See also:

- np.fmin Element-wise minimum of two arrays, ignores NaNs.
- np.maximum Element-wise maximum of two arrays, propagates NaNs.
- np.amax The maximum value of an array along a given axis, propagates NaNs.
- np.nanmax The maximum value of an array along a given axis, ignores NaNs.
- np.minimum, np.amin, np.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

```python
>>> np.fmax([2, 3, 4], [1, 5, 2])  # doctest: +SKIP
array([2., 5., 4.])
```
>>> np.fmax(np.eye(2), [0.5, 2])  # doctest: +SKIP
array([[ 1., 2.],
       [ 0.5, 2.]])

>>> np.fmax([np.nan, 0, np.nan], [0, np.nan, np.nan])  # doctest: +SKIP
array([ 0., 0., NaN])

dask.array.fmin(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then 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.
- out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- y [ndarray or scalar] The minimum of x1 and x2, element-wise. This is a 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

```python
>>> np.fmin([2, 3, 4], [1, 5, 2])  # doctest: +SKIP
array([1, 3, 2])
```

```python
>>> np.fmin(np.eye(2), [0.5, 2])  # doctest: +SKIP
array([[ 0.5, 0.],
       [ 0., 1.]])
```

```python
>>> np.fmin([np.nan, 0, np.nan], [0, np.nan, np.nan])  # doctest: +SKIP
array([ 0., 0., NaN])
```

dask.array.fmod(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [array_like] The remainder of the division of x1 by x2. This is a scalar if both x1 and x2 are scalars.

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)  # doctest: +SKIP
array([-1, 0, -1, 1, 0, 1])
>>> np.remainder([-3, -2, -1, 1, 2, 3], 2)  # doctest: +SKIP
array([1, 0, 1, 1, 0, 1])
>>> np.fmod([5, 3], [2, 2.])  # doctest: +SKIP
array([  1.,   1.])
>>> a = np.arange(-3, 3).reshape(3, 2)  # doctest: +SKIP
>>> a
array([[-3, -2],
       [-1,  0],
       [ 1,  2]])
>>> np.fmod(a, [2,2])  # doctest: +SKIP
array([[-1,  0],
       [-1,  0],
       [ 1,  0]])

dask.array.frexp(x[, out1, out2], [out=(None, None)], *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature, extobj)

Decompose the elements of x into mantissa and twos exponent.

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

Parameters

- **x** [array_like] Array of numbers to be decomposed.
- **out1** [ndarray, optional] Output array for the mantissa. Must have the same shape as x.
- **out2** [ndarray, optional] Output array for the exponent. Must have the same shape as x.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwags For other keyword-only arguments, see the ufunc docs.

Returns

- **mantissa** [ndarray] Floating values between -1 and 1. This is a scalar if x is a scalar.
- **exponent** [ndarray] Integer exponents of 2. This is a scalar if x is a scalar.

See also:

- **ldexp** Compute \( y = x_1 \times 2^{x_2} \), the inverse of frexp.

Notes

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

```python
>>> x = np.arange(9)  # doctest: +SKIP
>>> y1, y2 = np.frexp(x)  # doctest: +SKIP
>>> y1  # doctest: +SKIP
array([ 0. , 0.5 , 0.5 , 0.75 , 0.5 , 0.625, 0.75 , 0.875, 0.5 ])
>>> y2  # doctest: +SKIP
array([0, 1, 2, 2, 3, 3, 3, 3, 4])
>>> y1 * 2**y2  # doctest: +SKIP
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 \( f_n(x, y, z) \) 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 would be \( array([[[0, 0], [1, 1]]]) \) and \( array([[0, 1], [0, 1]]) \).

- **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 not match the \( shape \) parameter.

See also:

- indices, meshgrid

Notes

Keywords other than \( dtype \) are passed to \( function \).

Examples

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

```python
>>> 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

   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.

See also:

   vectorize evaluates pyfunc over input arrays using broadcasting rules of numpy

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([b'012', b'036', b'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 The default, None, means np.array(fill_value).dtype.
   order [‘C’, ‘F’], optional] Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns

   out [ndarray] Array of fill_value with the given shape, dtype, and order.
See also:

**full_like** Return a new array with shape of input filled with value.

**empty** Return a new uninitialized array.

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

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

### Examples

```python
def main():
    print(np.full((2, 2), np.inf))
    print(np.full((2, 2), 10))

if __name__ == '__main__':
    main()
```

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.gradient(f, *varargs, **kwargs)

Return the gradient of an N-dimensional array.

The gradient is computed using second order accurate central differences in the interior points and either first or second order accurate one-sides (forward or backwards) differences at the boundaries. The returned gradient hence has the same shape as the input array.

**Parameters**

- **f** [array_like] An N-dimensional array containing samples of a scalar function.
**varargs** [list of scalar or array, optional] Spacing between f values. Default unitary spacing for all dimensions. Spacing can be specified using:

1. single scalar to specify a sample distance for all dimensions.
2. N scalars to specify a constant sample distance for each dimension. i.e. $dx$, $dy$, $dz$, …
3. N arrays to specify the coordinates of the values along each dimension of F. The length of the array must match the size of the corresponding dimension
4. Any combination of N scalars/arrays with the meaning of 2. and 3.

If **axis** is given, the number of varargs must equal the number of axes. Default: 1.

**edge_order** [{1, 2}, optional] Gradient is calculated using N-th order accurate differences at the boundaries. Default: 1.

New in version 1.9.1.

**axis** [None or int or tuple of ints, optional] Gradient is calculated only along the given axis or axes The default (axis = None) is to calculate the gradient for all the axes of the input array. axis may be negative, in which case it counts from the last to the first axis.

New in version 1.11.0.

**Returns**

**gradient** [ndarray or list of ndarray] A set of ndarrays (or a single ndarray if there is only one dimension) corresponding to the derivatives of f with respect to each dimension. Each derivative has the same shape as f.

**Notes**

Assuming that $f \in C^3$ (i.e., f has at least 3 continuous derivatives) and let $h_s$ be a non-homogeneous stepsize, we minimize the “consistency error” $\eta_i$ between the true gradient and its estimate from a linear combination of the neighboring grid-points:

$$\eta_i = f_i^{(1)} - [\alpha f(x_i) + \beta f(x_i + h_d) + \gamma f(x_i - h_s)]$$

By substituting $f(x_i + h_d)$ and $f(x_i - h_s)$ with their Taylor series expansion, this translates into solving the following linear system:

$$\begin{align*}
\alpha + \beta + \gamma &= 0 \\
\beta h_d - \gamma h_s &= 1 \\
\beta h_d^2 + \gamma h_s^2 &= 0
\end{align*}$$

The resulting approximation of $f_i^{(1)}$ is the following:

$$f_i^{(1)} = \frac{h_s^2 f(x_i + h_d) + (h_d^2 - h_s^2) f(x_i) - h_d^2 f(x_i - h_s)}{h_s h_d (h_d + h_s)} + \mathcal{O}\left(\frac{h_d h_s^2 + h_s h_d^2}{h_d + h_s}\right)$$

It is worth noting that if $h_s = h_d$ (i.e., data are evenly spaced) we find the standard second order approximation:

$$f_i^{(1)} = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + \mathcal{O}(h^2)$$

With a similar procedure the forward/backward approximations used for boundaries can be derived.

**References**

[1], [2], [3]
Examples

```python
>>> f = np.array([1, 2, 4, 7, 11, 16], dtype=float)
>>> np.gradient(f)
array([ 1. , 1.5, 2.5, 3.5, 4.5, 5. ])
```

Spacing can be also specified with an array that represents the coordinates of the values \( F \) along the dimensions. For instance a uniform spacing:

```python
>>> x = np.arange(f.size)
>>> np.gradient(f, x)
array([ 1. , 1.5, 2.5, 3.5, 4.5, 5. ])
```

Or a non uniform one:

```python
>>> x = np.array([0., 1., 1.5, 3.5, 4., 6.], dtype=float)
>>> np.gradient(f, x)
array([ 1. , 3. , 3.5, 6.7, 6.9, 2.5])
```

For two dimensional arrays, the return will be two arrays ordered by axis. In this example the first array stands for the gradient in rows and the second one in columns direction:

```python
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]], dtype=float))
[array([[ 2., 2., -1.],
       [ 2., 2., -1.]]), array([[ 2. , 2. , 2. ],
       [ 1. , 1. , 1. ]])]
```

In this example the spacing is also specified: uniform for axis=0 and non uniform for axis=1

```python
>>> dx = 2.
>>> y = [1., 1.5, 3.5]
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]], dtype=float), dx, y)
[array([[ 1. , 1. , -0.5],
       [ 2. , 1.7, 0.5]]), array([[ 2. , 2. , 2. ],
       [ 2. , 1.7, 0.5]])]
```

It is possible to specify how boundaries are treated using `edge_order`

```python
>>> x = np.array([0, 1, 2, 3, 4])
>>> f = x**2
>>> np.gradient(f, edge_order=1)
array([ 1., 2., 4., 6., 7.])
>>> np.gradient(f, edge_order=2)
array([ 2., 2., 2. , 2. , 2. ])  # Note: Boundary treatment for edge_order=2
```

The `axis` keyword can be used to specify a subset of axes of which the gradient is calculated

```python
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]], dtype=float), axis=0)
array([[ 2., 2., -1.],
       [ 2., 2., -1.]])
```

`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:

```python
d >>> import dask.array as da
d >>> import numpy as np
d >>> 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:

```python
d >>> 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).

This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis. Rebuilds arrays divided by hsplit.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions concatenate, stack and block provide more general stacking and concatenation operations.

Parameters

tup [sequence of ndarrays] The arrays must have the same shape along all but the second axis, except 1-D arrays which can be any length.

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.
block Assemble arrays from blocks.
Examples

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

```python
da = 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=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

**Returns**

- **z** [ndarray] The hypotenuse of the triangle(s). This is a scalar if both \( x1 \) and \( x2 \) are scalars.

Examples

```python
>>> 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:

```python
>>> 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 complex argument.

**Parameters**

- **val** [array_like] Input array.

**Returns**
out [ndarray or scalar] The imaginary component of the complex argument. If val is real, the
type of val is used for the output. If val has complex elements, the returned type is float.

See also:
real, angle, real_if_close

Examples

```python
>>> a = np.array([1+2j, 3+4j, 5+6j])  # doctest: +SKIP
>>> a.imag  # doctest: +SKIP
array([ 2., 4., 6.])
>>> a.imag = np.array([8, 10, 12])  # doctest: +SKIP
>>> a  # doctest: +SKIP
array([ 1. +8.j, 3.+10.j, 5.+12.j])
>>> np.imag(1 + 1j)  # doctest: +SKIP
1.0
```

dask.array.indices(dimensions, dtype=<class 'int'>, chunks='auto')
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**

```python
>>> 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:**

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

```python
>>> 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])
```

```python
>>> 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.invert(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj ])

Compute bit-wise inversion, or bit-wise NOT, element-wise.

Computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ~.

For signed integer inputs, the two’s complement is returned. In a two’s-complement system negative numbers are represented by the two’s complement of the absolute value. This is the most common method of representing signed integers on computers [1]. A N-bit two’s-complement system can represent every integer in the range $-2^{N-1} + 1$ to $+2^{N-1} - 1$.

Parameters

- x (array_like) Only integer and boolean types are handled.
- out (ndarray, None, or tuple of ndarray and None, optional) A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- where (array_like, optional) Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- out (ndarray or scalar) Result. This is a scalar if x is a scalar.

See also:

- bitwise_and, bitwise_or, bitwise_xor, logical_not
- binary_repr Return the binary representation of the input number as a string.

Notes

bitwise_not is an alias for invert:

```python
>>> np.bitwise_not is np.invert  # doctest: +SKIP
True
```

References

[1]

Examples

We’ve seen that 13 is represented by 00001101. The invert or bit-wise NOT of 13 is then:

```python
>>> np.invert(np.array([13], dtype=uint8))  # doctest: +SKIP
array([242], dtype=uint8)
>>> np.binary_repr(x, width=8)  # doctest: +SKIP
'00001101'
>>> np.binary_repr(242, width=8)  # doctest: +SKIP
'11110010'
```

The result depends on the bit-width:
>>> np.invert(np.array([13], dtype=uint16)) # doctest: +SKIP
array([65522], dtype=uint16)
>>> np.binary_repr(65522, width=16) # doctest: +SKIP
'1111111111110010'

When using signed integer types the result is the two's complement of the result for the unsigned type:

>>> np.invert(np.array([13], dtype=int8)) # doctest: +SKIP
array([-14], dtype=int8)
>>> np.binary_repr(-14, width=8) # doctest: +SKIP
'11110010'

Booleans are accepted as well:

>>> np.invert(array([True, False])) # doctest: +SKIP
array([False, True])

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 \((rtol \times \text{abs}(b))\) and the absolute difference \(atol\) are added together to compare against the absolute difference between \(a\) and \(b\).

**Warning:** The default \(atol\) is not appropriate for comparing numbers that are much smaller than one (see Notes).

**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.

\[
\text{absolute}(a - b) \leq (\text{atol} + \text{rtol} \times \text{absolute}(b))
\]
Unlike the built-in `math.isclose`, the above equation is not symmetric in \(a\) and \(b\) – it assumes \(b\) is the reference value – so that `isclose(a, b)` might be different from `isclose(b, a)`. Furthermore, the default value of atol is not zero, and is used to determine what small values should be considered close to zero. The default value is appropriate for expected values of order unity: if the expected values are significantly smaller than one, it can result in false positives. \(atol\) should be carefully selected for the use case at hand. A zero value for \(atol\) will result in `False` if either \(a\) or \(b\) is zero.

**Examples**

```python
>>> 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])
>>> np.isclose([1e-8, 1e-7], [0.0, 0.0])
array([ True, False, False, False, False, True])
>>> np.isclose([1e-10, 1e-10], [1e-20, 0.999999e-10], atol=0.0)
array([False, True], dtype=bool)
```

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**


**See also:**

- `isreal`
- `iscoplexobj` Return True if x is a complex type or an array of complex numbers.

**Examples**

```python
>>> np.iscomplex([1+1j, 1+0j, 4.5, 3, 2, 2j])  # doctest: +SKIP
array([ True, False, False, False, False, True])
```

dask.array.isfinite(x, /, out=None, *, where=True, casting=`same_kind`, order=`K`, dtype=None, subok=True, signature=None, extobj=None)

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, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray, bool] True where $x$ is not positive infinity, negative infinity, or NaN; false otherwise.

This is a scalar if $x$ is a scalar.

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

```python
def doctest_example
>>> np.isfinite(1)  # doctest: +SKIP
True
>>> np.isfinite(0)  # doctest: +SKIP
True
>>> np.isfinite(np.nan)  # doctest: +SKIP
False
>>> np.isfinite(np.inf)  # doctest: +SKIP
False
>>> np.isfinite(np.NINF)  # doctest: +SKIP
False
>>> np.isfinite([np.log(-1.),1.,np.log(0)])  # doctest: +SKIP
array([False, True, False])
```

```python
def doctest_example
>>> x = np.array([-np.inf, 0., np.inf])  # doctest: +SKIP
>>> y = np.array([2, 2, 2])  # doctest: +SKIP
>>> np.isfinite(x, y)  # doctest: +SKIP
array([0, 1, 0])
```

dask.array.isin(element, test_elements, assume_unique=False, invert=False)
Calculates element in test_elements, broadcasting over element only. Returns a boolean array of the same shape as element that is True where an element of element is in test_elements and False otherwise.

Parameters
element  [array_like] Input array.

test_elements  [array_like] The values against which to test each value of element. This argument is flattened if it is an array or array_like. See notes for behavior with non-array-like parameters.

assume_unique  [bool, optional] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

invert  [bool, optional] If True, the values in the returned array are inverted, as if calculating element not in test_elements. Default is False. np.isin(a, b, invert=True) is equivalent to (but faster than) np.invert(np.isin(a, b)).

Returns

isin  [ndarray, bool] Has the same shape as element. The values element[isin] are in test_elements.

See also:

in1d Flattened version of this function.

numpy.lib.arraysetops Module with a number of other functions for performing set operations on arrays.

Notes

isin is an element-wise function version of the python keyword in. isin(a, b) is roughly equivalent to np.array([item in b for item in a]) if a and b are 1-D sequences.

element and test_elements are converted to arrays if they are not already. If test_elements is a set (or other non-sequence collection) it will be converted to an object array with one element, rather than an array of the values contained in test_elements. This is a consequence of the array constructor’s way of handling non-sequence collections. Converting the set to a list usually gives the desired behavior.

New in version 1.13.0.

Examples

```python
>>> element = 2*np.arange(4).reshape((2, 2))
>>> element
array([[0, 2],
       [4, 6]])
>>> test_elements = [1, 2, 4, 8]
>>> mask = np.isin(element, test_elements)
>>> mask
array([[False,  True],
       [ True, False]])
>>> element[mask]
array([2, 4])
```

The indices of the matched values can be obtained with nonzero:

```python
>>> np.nonzero(mask)
(array([0, 1]), array([1, 0]))
```

The test can also be inverted:
```python
>>> mask = np.isin(element, test_elements, invert=True)
>>> mask
array([[ True, False],
        [False, True]])
>>> element[mask]
array([0, 6])
```

Because of how `array` handles sets, the following does not work as expected:

```python
>>> test_set = {1, 2, 4, 8}
>>> np.isin(element, test_set)
array([[False, False],
       [False, False]])
```

Casting the set to a list gives the expected result:

```python
>>> np.isin(element, list(test_set))
array([[False, True],
       [True, False]])
```

dask.array.isinf(x, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, [signature, extobj])

Test element-wise for positive or negative infinity.

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

**Parameters**

- **x** [array_like] Input values
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

**Returns**

- **y** [bool (scalar) or boolean ndarray] True where `x` is positive or negative infinity, false otherwise. This is a scalar if `x` is a scalar.

**See also:**

- isneginf, isposinf, isnan, isfinite

**Notes**

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is supplied when the first argument is a scalar, or if the first and second arguments have different shapes.
Examples

```python
>>> np.isinf(np.inf)  # doctest: +SKIP
True
>>> np.isinf(np.nan)  # doctest: +SKIP
False
>>> np.isinf(np.NINF)  # doctest: +SKIP
True
>>> np.isinf([np.inf, -np.inf, 1.0, np.nan])  # doctest: +SKIP
array([ True,  True, False, False])
>>> x = np.array([-np.inf, 0., np.inf])
# doctest: +SKIP
>>> y = np.array([2, 2, 2])  # doctest: +SKIP
>>> np.isinf(x, y)  # doctest: +SKIP
array([1, 0, 1])
>>> y  # doctest: +SKIP
array([1, 0, 1])
```

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

Test element-wise for negative infinity, return result as bool array.

Parameters

- `x` [array_like] The input array.
- `out` [array_like, optional] A boolean array with the same shape and type as `x` to store the result.

Returns

- `out` [ndarray] A boolean array with the same dimensions as the input. If second argument is not supplied then a numpy boolean array is returned with values True where the corresponding element of the input is negative infinity and values False where the element of the input is not negative infinity.

If a second argument is 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. The return value `out` is then a reference to that array.

See also:

isinf, isposinf, isnan, isfinite

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is also supplied when `x` is a scalar input, if first and second arguments have different shapes, or if the first argument has complex values.

Examples

```python
>>> np.isneginf(np.NINF)  # doctest: +SKIP
array(True, dtype=bool)
>>> np.isneginf(np.inf)  # doctest: +SKIP
array(False, dtype=bool)
>>> np.isneginf(np.PINF)  # doctest: +SKIP
array(False, dtype=bool)
```

(continues on next page)
>>> np.isneginf([-np.inf, 0., np.inf])  # doctest: +SKIP
array([ True, False, False])

>>> x = np.array([-np.inf, 0., np.inf])  # doctest: +SKIP
>>> y = np.array([2, 2, 2])  # doctest: +SKIP
>>> np.isneginf(x, y)  # doctest: +SKIP
array([1, 0, 0])

>>> y  # doctest: +SKIP
array([1, 0, 0])

dask.array.isnan(x, I, out=None, *, where=True, casting='same_kind', order='K', subok=True, signature=None, extobj=None)
Test element-wise for NaN and return result as a boolean array.

Parameters

- x [array_like] Input array.
- out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- y [ndarray or bool] True where x is NaN, false otherwise. This is a scalar if x is a scalar.

See also:

isinf, isneginf, isposinf, isnan, isfinite, isinf, isnull
pandas.isnull for dask arrays

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)  # doctest: +SKIP
True
>>> np.isnan(np.inf)  # doctest: +SKIP
False
>>> np.isnan([np.log(-1.),1.,np.log(0)])  # doctest: +SKIP
array([ True, False, False])

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

dask.array.isposinf(*args, **kwargs)
Test element-wise for positive infinity, return result as bool array.

Parameters
x [array_like] The input array.

y [array_like, optional] A boolean array with the same shape as x to store the result.

Returns

out [ndarray] A boolean array with the same dimensions as the input. If second argument is not supplied then a boolean array is returned with values True where the corresponding element of the input is positive infinity and values False where the element of the input is not positive infinity.

If a second argument is 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. The return value out is then a reference to that array.

See also:

isinf, isneginf, isfinite, isnan

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is also supplied when x is a scalar input, if first and second arguments have different shapes, or if the first argument has complex values

Examples

```python
>>> np.isposinf(np.PINF)  # doctest: +SKIP
array(True, dtype=bool)
>>> np.isposinf(np.inf)  # doctest: +SKIP
array(True, dtype=bool)
>>> np.isposinf(np.NINF)  # doctest: +SKIP
array(False, dtype=bool)
>>> np.isposinf([-np.inf, 0., np.inf])  # doctest: +SKIP
array([False, False, True])
```

```python
>>> x = np.array([-np.inf, 0., np.inf])  # doctest: +SKIP
>>> y = np.array([2, 2, 2])  # doctest: +SKIP
>>> np.isposinf(x, y)  # doctest: +SKIP
array([0, 0, 1])
>>> y  # doctest: +SKIP
array([0, 0, 1])
```

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**

```python
>>> np.isreal([1+1j, 1+0j, 4.5, 3, 2, 2j])  # doctest: +SKIP
array([False, True, True, True, True, False])
```

dask.array.ldexp(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj ])

Returns \( x_1 \times 2^{x_2} \), element-wise.

The mantissas \( x_1 \) and twos exponents \( x_2 \) are used to construct floating point numbers \( x_1 \times 2^{x_2} \).

**Parameters**

- \( x_1 \) [array_like] Array of multipliers.
- \( x_2 \) [array_like, int] Array of twos exponents.
- \( out \) [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- \( where \) [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

**Returns**

- \( y \) [ndarray or scalar] The result of \( x_1 \times 2^{x_2} \). This is a scalar if both \( x_1 \) and \( x_2 \) are scalars.

**See also:**

frexp Return \((y_1, y_2)\) from \(x = y_1 \times 2^{y_2}\), 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 \( x_1 \times 2^{x_2} \).

**Examples**

```python
>>> np.ldexp(5, np.arange(4))  # doctest: +SKIP
array([ 5., 10., 20., 40.], dtype=float32)

>>> x = np.arange(6)  # doctest: +SKIP

>>> np.ldexp(*np.frexp(x))  # doctest: +SKIP
array([ 0., 1., 2., 3., 4., 5.])
```

dask.array.linspace(start, stop, num=50, endpoint=True, retstep=False, chunks='auto', dtype=None)

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

**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. Default is 50.
endpoint [bool, optional] If True, stop is the last sample. Otherwise, it is not included. Default is True.
retstep [bool, optional] If True, return (samples, step), where step is the spacing between samples. Default is False.
chunks [int] The number of samples on each block. Note that the last block will have fewer samples if num % blocksize != 0
dtype [dtype, optional] The type of the output array.

Returns
samples [dask array] Only returned if retstep is True. Size of spacing between samples.

See also:
dask.array.arange
dask.array.log

Parameters
x [array_like] Input value.
out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns
y [ndarray] The natural logarithm of x, element-wise. This is a scalar if x is a scalar.

See also:
log10, log2, log1p, emath.log

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

For real-valued input data types, log always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.
For complex-valued input, \( \log \) is a complex analytical function that has a branch cut \([-\infty, 0] \) and is continuous from above on it. \( \log \) handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

**References**

[1], [2]

**Examples**

```python
>>> np.log([1, np.e, np.e**2, 0])  # doctest: +SKIP
array([ 0., 1., 2., -Inf])
```

dask.array.log10(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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

**Parameters**

- **x** [array_like] Input values.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwags For other keyword-only arguments, see the ufunc docs.

**Returns**

- **y** [ndarray] The logarithm to the base 10 of \( x \), element-wise. NaNs are returned where \( x \) is negative. This is a scalar if \( x \) is a scalar.

**See also:**

dask.array.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, \( \log 10 \) 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, \( \log 10 \) is a complex analytical function that has a branch cut \([-\infty, 0] \) and is continuous from above on it. \( \log 10 \) handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

**References**

[1], [2]
Examples

```python
>>> np.log10([1e-15, -3.])  # doctest: +SKIP
array([-15., NaN])
```

dask.array.log1p(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True)

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

Calculates $\log(1 + x)$.

Parameters

- **x** [array_like] Input values.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [ndarray] Natural logarithm of $1 + x$, element-wise. This is a scalar if $x$ is a scalar.

See also:

expm1 $\exp(x) - 1$, the inverse of log1p.

Notes

For real-valued input, log1p is accurate also for $x$ so small that $1 + x == 1$ in floating-point accuracy.

Logarithm is a multivalued function: for each $x$ there is an infinite number of $z$ such that $\exp(z) = 1 + x$. The convention is to return the $z$ whose imaginary part lies in $[-\pi, \pi]$.

For real-valued input data types, log1p always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, log1p is a complex analytical function that has a branch cut $[-\infty, -1]$ and is continuous from above on it. log1p handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[1], [2]

Examples

```python
>>> np.log1p(1e-99)  # doctest: +SKIP
1e-99
>>> np.log(1 + 1e-99)  # doctest: +SKIP
0.0
```
dask.array.log2 (x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature, extobj)

Base-2 logarithm of x.

Parameters

- x [array_like] Input values.
- out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- y [ndarray] Base-2 logarithm of x. This is a scalar if x is a scalar.

See also:

log, log10, log1p, emath.log2

Notes

New in version 1.3.0.

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

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

For complex-valued input, log2 is a complex analytical function that has a branch cut [-inf, 0] and is continuous from above on it. log2 handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

Examples

```python
>>> x = np.array([0, 1, 2, 2**4])  # doctest: +SKIP
>>> np.log2(x)  # doctest: +SKIP
array([-Inf,  0.,  1.,  4.])
```

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

dask.array.logaddexp (x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature, extobj)

Logarithm of the sum of exponentiations of the inputs.

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.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

result [ndarray] Logarithm of \(\exp(x1) + \exp(x2)\). This is a scalar if both \(x1\) and \(x2\) are scalars.

See also:

*logaddexp2* Logarithm of the sum of exponentiations of inputs in base 2.

Notes

New in version 1.3.0.

Examples

```python
>>> prob1 = np.log(1e-50)  # doctest: +SKIP
>>> prob2 = np.log(2.5e-50)  # doctest: +SKIP
>>> prob12 = np.logaddexp(prob1, prob2)  # doctest: +SKIP
>>> prob12  # doctest: +SKIP
-113.87649168120691
>>> np.exp(prob12)  # doctest: +SKIP
3.5000000000000057e-50
```

dask.array.logaddexp2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns
result  [ndarray] Base-2 logarithm of \(2^{x_1} + 2^{x_2}\). This is a scalar if both \(x_1\) and \(x_2\) are scalars.

See also:

logaddexp  Logarithm of the sum of exponentiations of the inputs.

Notes

New in version 1.3.0.

Examples

```python
>>> prob1 = np.log2(1e-50)  # doctest: +SKIP
>>> prob2 = np.log2(2.5e-50)  # doctest: +SKIP
>>> prob12 = np.logaddexp2(prob1, prob2)  # doctest: +SKIP
>>> prob1, prob2, prob12
(-166.09640474436813, -164.77447664948076, -164.28904982231052)
>>> 2**prob12  # doctest: +SKIP
3.4999999999999914e-50
```

dask.array.logical_and(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

Compute the truth value of \(x_1\) AND \(x_2\) element-wise.

Parameters

- **x1** [array_like] Input arrays. \(x_1\) and \(x_2\) must be of the same shape.
- **x2** [array_like] Input arrays. \(x_1\) and \(x_2\) must be of the same shape.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs** For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [ndarray or bool] Boolean result with the same shape as \(x_1\) and \(x_2\) of the logical AND operation on corresponding elements of \(x_1\) and \(x_2\). This is a scalar if both \(x_1\) and \(x_2\) are scalars.

See also:

logical_or, logical_not, logical_xor, bitwise_and

Examples

```python
>>> np.logical_and(True, False)  # doctest: +SKIP
False
>>> np.logical_and([True, False], [False, False])  # doctest: +SKIP
array([[False, False]])
```
```python
>>> x = np.arange(5)  # doctest: +SKIP
>>> np.logical_and(x>1, x<4)  # doctest: +SKIP
array([False, False, True, True, False])
```

dask.array.logical_not(x, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

Compute the truth value of NOT x element-wise.

**Parameters**

- **x** [array_like] Logical NOT is applied to the elements of x.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwarg**s For other keyword-only arguments, see the ufunc docs.

**Returns**

- **y** [bool or ndarray of bool] Boolean result with the same shape as x of the NOT operation on elements of x. This is a scalar if x is a scalar.

**See also:**

logical_and, logical_or, logical_xor

**Examples**

```python
>>> np.logical_not(3)  # doctest: +SKIP
False
>>> np.logical_not([True, False, 0, 1])  # doctest: +SKIP
array([False, True, True, False])
```

dask.array.logical_or(x1, x2, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwarg**s For other keyword-only arguments, see the ufunc docs.
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. This is a scalar if both x1 and x2 are scalars.

See also:

logical_and, logical_not, logical_xor, bitwise_or

Examples

```python
>>> np.logical_or(True, False)  # doctest: +SKIP
True
>>> np.logical_or([True, False], [False, False])  # doctest: +SKIP
array([ True, False])
```

dask.array.logical_xor(x1, x2, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

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. This is a scalar if both x1 and x2 are scalars.

See also:

logical_and, logical_or, logical_not, bitwise_xor

Examples

```python
>>> np.logical_xor(True, False)  # doctest: +SKIP
True
>>> np.logical_xor([True, False, False], [False, False, False])  # doctest: +SKIP
array([False, True, True])
```
>>> x = np.arange(5)  # doctest: +SKIP
>>> np.logical_xor(x < 1, x > 3)  # doctest: +SKIP
array([ True, False, False, False,  True])

Simple example showing support of broadcasting

>>> np.logical_xor(0, np.eye(2))  # doctest: +SKIP
array([[ True, False],
       [False,  True]])

dask.array.matumul(a, b)
dask.array.max(a, axis=None, out=None, keepdims=<no value>, initial=<no value>)

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.
New in version 1.7.0.
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 input array.
If the default value is passed, then keepdims will not be passed through to the amax method of sub-classes of ndarray, however any non-default value will be. If the sub-class’ method does not implement keepdims any exceptions will be raised.
initial [scalar, optional] The minimum value of an output element. Must be present to allow computation on empty slice. See ~numpy.ufunc.reduce for details.
New in version 1.15.0.

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

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

```python
>>> b = np.arange(5, dtype=float)
>>> np.amax(b)  # Maximum of the flattened array
nan
>>> np.nanmax(b)  # Maximum of the flattened array
4.0
```

You can use an initial value to compute the maximum of an empty slice, or to initialize it to a different value:

```python
>>> np.max([-50], [10], axis=-1, initial=0)
array([ 0, 10])
```

Notice that the initial value is used as one of the elements for which the maximum is determined, unlike for the default argument Python’s max function, which is only used for empty iterables.

```python
>>> np.max([5], initial=6)
6
>>> max([5], default=6)
5
```

dask.array.maximum(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj ])

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.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or
None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The maximum of x1 and x2, element-wise. This is a scalar if both x1 and x2 are scalars.

See also:

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

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

```python
>>> np.maximum([2, 3, 4], [1, 5, 2])  # doctest: +SKIP
array([2, 5, 4])

>>> np.maximum(np.eye(2), [0.5, 2])  # broadcasting  # doctest: +SKIP
array([[ 1. , 2. ],
       [ 0.5, 2. ]])

>>> np.maximum([np.nan, 0, np.nan], [0, np.nan, np.nan])  # doctest: +SKIP
array([ NaN, 0. , NaN])

>>> np.maximum(np.Inf, 1)  # doctest: +SKIP
-inf
```

dask.array.mean (a, axis=None, dtype=None, out=None, keepdims=<no value>)

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.

New in version 1.7.0.
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 input array.

If the default value is passed, then `keepdims` will not be passed through to the `mean` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class’ method does not implement `keepdims` any exceptions will be raised.

**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.

By default, `float16` results are computed using `float32` intermediates for extra precision.

**Examples**

```python
>>> 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:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924
```

Computing the mean in `float64` is more accurate:
```python
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

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.mgrid 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:

```python
xv, yv = np.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 = np.meshgrid(x, y, sparse=False, indexing='xy')
for i in range(nx):
```

(continues on next page)
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

```python
>>> nx, ny = (3, 2)
>>> x = np.linspace(0, 1, nx)
>>> y = np.linspace(0, 1, ny)
>>> xv, yv = np.meshgrid(x, y)
>>> xv
array([[ 0. , 0.5, 1. ],
       [ 0. , 0.5, 1. ]])
>>> yv
array([[ 0., 0., 0.],
       [ 1., 1., 1.]])
>>> xv, yv = np.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.
```

```python
>>> import matplotlib.pyplot as plt
>>> x = np.arange(-5, 5, 0.1)
>>> y = np.arange(-5, 5, 0.1)
>>> xx, yy = np.meshgrid(x, y, sparse=True)
>>> z = np.sin(xx**2 + yy**2) / (xx**2 + yy**2)
>>> h = plt.contourf(x,y,z)
>>> plt.show()
```

dask.array.min(a, axis=None, out=None, keepdims=<no value>, initial=<no value>)

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.
  
  New in version 1.7.0.
  
  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 input array.
If the default value is passed, then `keepdims` will not be passed through to the `amin` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class’ method does not implement `keepdims` any exceptions will be raised.

```
initial [scalar, optional] The maximum value of an output element. Must be present to allow computation on empty slice. See ~numpy.ufunc.reduce for details.
```

New in version 1.15.0.

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

```python
>>> 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])
```

```python
>>> b = np.arange(5, dtype=float)
>>> np.amin(b)
nan
>>> np.nanmin(b)
0.0
```

```python
>>> np.min([[[-50], [10]], axis=-1, initial=0)
array([-50, 0])
```

4.7. Array
Notice that the initial value is used as one of the elements for which the minimum is determined, unlike for the
default argument Python’s max function, which is only used for empty iterables.

Notice that this isn’t the same as Python’s default argument.

```python
>>> np.min([6], initial=5)
5
>>> min([6], default=5)
6
```

dask.array.minimum(x1, x2, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj ])
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.

- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result
  is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or
  None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument)
  must have length equal to the number of outputs.

- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position,
  values of False indicate to leave the value in the output alone.

- ****kwags For other keyword-only arguments, see the ufunc docs.

**Returns**

- **y** [ndarray or scalar] The minimum of x1 and x2, element-wise. This is a scalar if both x1 and
  x2 are scalars.

**See also:**

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

```python
>>> np.minimum([2, 3, 4], [1, 5, 2])  # doctest: +SKIP
array([1, 3, 2])

>>> np.minimum(np.eye(2), [0.5, 2])  # broadcasting  # doctest: +SKIP
array([[ 0.5, 0.0],
       [ 0.0, 1.0]])

>>> np.minimum([np.nan, 0, np.nan], [0, np.nan, np.nan])  # doctest: +SKIP
array([ NaN, NaN, NaN])

>>> np.minimum(-np.inf, 1)  # doctest: +SKIP
-inf
```

dask.array.modf(x[, out1, out2], /[, out=(None, None)], *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

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.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **y1** [ndarray] Fractional part of x. This is a scalar if x is a scalar.
- **y2** [ndarray] Integral part of x. This is a scalar if x is a scalar.

See also:

divmod divmod(x, 1) is equivalent to modf with the return values switched, except it always has a positive remainder.

Notes

For integer input the return values are floats.

Examples

```python
>>> np.modf([0, 3.5])  # doctest: +SKIP
(array([0.0, 0.5]), array([0., 3.]))

>>> np.modf(-0.5)  # doctest: +SKIP
(-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

```python
>>> 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.],
       [ 3.,  2.]])
>>> 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

```python
>>> 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=<no value>*)

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, tuple of int, None], optional] Axis or axes 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*.

If the value is anything but the default, then *keepdims* will be passed through to the *max* method of sub-classes of *ndarray*. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

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**

```python
>>> 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:
```python
>>> 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=<no value>)

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, tuple of int, None}, optional] Axis or axes 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 `a`.

If the value is anything but the default, then `keepdims` will be passed through to the mean or sum methods of sub-classes of ndarray. If the sub-classes methods does not implement `keepdims` any exceptions will be raised.

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

```python
>>> 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=<no value>)

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, tuple of int, None], optional] Axis or axes 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`.

If the value is anything but the default, then `keepdims` will be passed through to the `min` method of sub-classes of `ndarray`. If the sub-classes methods does not implement `keepdims` any exceptions will be raised. 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

```python
>>> 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:

```python
>>> 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=None)

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

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 product is desired. If a is not an array, a conversion is attempted.
- **axis** [{int, tuple of int, None}, optional] Axis or axes 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

- **nanprod** [ndarray] A new array holding the result is returned unless out is specified, in which case it is returned.

See also:
**numpy.prod**  Product across array propagating NaNs.

**isnan**  Show which elements are NaN.

### Examples

```python
>>> 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**  
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, tuple of int, None], optional]  Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.
- **dtype**  [dtype, optional]  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.
- **out**  [ndarray, optional]  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.
- **ddof**  [int, optional]  Means Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of non-NaN elements. By default \( ddof \) is zero.
- **keepdims**  [bool, optional]  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original \( a \).

If this value is anything but the default it is passed through as-is to the relevant functions of the sub-classes. If these functions do not have a \( keepdims \) kwarg, a RuntimeError will be raised.

**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:

\texttt{var, mean, std, nanvar, nanmean}

\texttt{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 $N - ddof$ 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, $\text{std}$ takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the $\text{std}$ 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 $\text{dtype}$ keyword can alleviate this issue.

Examples

```python
>>> 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=<no value>)

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

In NumPy versions <= 1.9.0 Nan is returned for slices that are all-NaN or empty. In later versions zero is returned.

Parameters

- \texttt{a} [array_like] Array containing numbers whose sum is desired. If \texttt{a} is not an array, a conversion is attempted.
- \texttt{axis} [\{int, tuple of int, None\}, optional] Axis or axes along which the sum is computed. The default is to compute the sum of the flattened array.
- \texttt{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 \texttt{a} is used. An exception is when \texttt{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 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.

If the value is anything but the default, then keepdims will be passed through to the mean or sum methods of sub-classes of ndarray. If the sub-classes methods does not implement keepdims any exceptions will be raised.

New in version 1.8.0.

Returns

nansum [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.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).

Examples

```python
>>> np.nansum([1, 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=<no value>)

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, tuple of int, None], optional Axis or axes 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 \( \text{ddof} = 0 \).
- **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*.

Returns

- **variance** [ndarray, see dtype parameter above] If *out* is None, return a new array containing the variance, otherwise return a reference to the output array. If *ddof* is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

See also:

- **std** Standard deviation
- **mean** Average
- **var** Variance while not ignoring NaNs
- **nanstd**, **nanmean**
- **numpy.doc.ufuncs** Section “Output arguments”

Notes

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - \text{mean}(x))^2) \).

The mean is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator of the variance of a hypothetical infinite population. \( \text{ddof}=0 \) 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.

For this function to work on sub-classes of ndarray, they must define sum with the kwarg keepdims.

### Examples

```python
>>> 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.nan_to_num(*args, **kwargs)
Replace NaN with zero and infinity with large finite numbers.

If x is inexact, NaN is replaced by zero, and infinity and -infinity replaced by the respectively largest and most negative finite floating point values representable by x.dtype.

For complex dtypes, the above is applied to each of the real and imaginary components of x separately.

If x is not inexact, then no replacements are made.

**Parameters**

- x [scalar or array_like] Input data.
- copy [bool, optional] Whether to create a copy of x (True) or to replace values in-place (False). The in-place operation only occurs if casting to an array does not require a copy. Default is True.

**Returns**

- out [ndarray] x, with the non-finite values replaced. If copy is False, this may be x itself.

**See also:**

- isinf Shows which elements are positive or negative infinity.
- isneginf Shows which elements are negative infinity.
- isposinf Shows which elements are positive infinity.
- isnan Shows which elements are Not a Number (NaN).
- isfinite Shows which elements are finite (not NaN, not infinity)

**Notes**

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.
Examples

```python
>>> np.nan_to_num(np.inf) # doctest: +SKIP
1.7976931348623157e+308
>>> np.nan_to_num(-np.inf) # doctest: +SKIP
-1.7976931348623157e+308
>>> np.nan_to_num(np.nan) # doctest: +SKIP
0.0
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128]) # doctest: +SKIP
>>> np.nan_to_num(x) # doctest: +SKIP
array([ 1.79769313e+308, -1.79769313e+308, 0.00000000e+000,
       -1.28000000e+002, 1.28000000e+002])
```

dask.array.nextafter(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or **None**, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **out** [ndarray or scalar] The next representable values of **x1** in the direction of **x2**. This is a scalar if both **x1** and **x2** are scalars.

Examples

```python
>>> eps = np.finfo(np.float64).eps # doctest: +SKIP
>>> np.nextafter(1, 2) == eps + 1 # doctest: +SKIP
True
>>> np.nextafter([1, 2], [2, 1]) == [eps + 1, 2 - eps] # doctest: +SKIP
array([[ True,  True]])
```

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

\[
\text{transpose(nonzero(a))}
\]

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

```python
>>> x = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
>>> x
array([[3, 0, 0],
       [0, 4, 0],
       [5, 6, 0]])
>>> np.nonzero(x)
(array([0, 1, 2, 2]), array([0, 1, 0, 1]))
>>> x[np.nonzero(x)]
array([3, 4, 5, 6])
```

Using this result to index \(a\) is equivalent to using the mask directly:

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]])
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```
nonzero can also be called as a method of the array.

```python
>>> (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, default: C] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**

- **out** [ndarray] Array of ones with the given shape, dtype, and order.

See also:

- **ones_like** Return an array of ones with shape and type of input.
- **empty** Return a new uninitialized array.
- **zeros** Return a new array setting values to zero.
- **full** Return a new array of given shape filled with value.

**Examples**

```python
>>> np.ones(5)
array([1., 1., 1., 1., 1.])

>>> np.ones((5,), dtype=int)
array([1, 1, 1, 1, 1])

>>> np.ones((2, 1))
array([[1.],
       [1.]])
```
```python
>>> 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.outer(a, b, out=None)

Compute the outer product of two vectors.

Given two vectors, \( a = [a_0, a_1, \ldots, a_M] \) and \( b = [b_0, b_1, \ldots, b_N] \), the outer product \([1]\) is:

\[
\begin{bmatrix}
  [a_0*b_0 & a_0*b_1 & \ldots & a_0*b_N ] \\
  [a_1*b_0 & \ldots ] \\
  [a_M*b_0 & a_M*b_N ]
\end{bmatrix}
\]

Parameters

- `a` [(M,) array_like] First input vector. Input is flattened if not already 1-dimensional.
- `b` [(N,) array_like] Second input vector. Input is flattened if not already 1-dimensional.
- `out` [(M, N) ndarray, optional] A location where the result is stored

New in version 1.9.0.

Returns

`out` [(M, N) ndarray] out[i, j] = a[i] * b[j]

See also:

inner
einsum einsum('i,j->ij', a.ravel(), b.ravel()) is the equivalent.
ufunc.outer A generalization to N dimensions and other operations. np.multiply.outer(a.ravel(), b.ravel()) is the equivalent.

References

[1]

Examples

Make a (very coarse) grid for computing a Mandelbrot set:

```python
>>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> rl
array([[[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]]])
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])
>>> grid = rl + im
>>> grid
array([[[-2.+2.j, -1.+2.j, 0.+2.j, 1.+2.j, 2.+2.j],
        [-2.+1.j, -1.+1.j, 0.+1.j, 1.+1.j, 2.+1.j],
        [-2.+0.j, -1.+0.j, 0.+0.j, 1.+0.j, 2.+0.j],
        [-2.-1.j, -1.-1.j, 0.-1.j, 1.-1.j, 2.-1.j],
        [-2.-2.j, -1.-2.j, 0.-2.j, 1.-2.j, 2.-2.j]]])
```

An example using a “vector” of letters:

```python
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([['a', 'aa', 'aaa'],
       ['b', 'bb', 'bbb'],
       ['c', 'cc', 'ccc']], dtype=object)
```

dask.array.pad(array, pad_width, mode, **kwargs)

Pads an array.

Parameters

array [array_like of rank N] Input array

pad_width [[sequence, array_like, int]] Number of values padded to the edges of each axis.

(mode, after), ... (before_N, after_N)) unique pad widths for each axis. (before, after,) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes.

mode [str or function] One of the following string values or a user supplied function.

‘constant’ Pads with a constant value.

‘edge’ Pads with the edge values of array.
‘linear_ramp’ Pads with the linear ramp between end_value and the array edge value.

‘maximum’ Pads with the maximum value of all or part of the vector along each axis.

‘mean’ Pads with the mean value of all or part of the vector along each axis.

‘median’ Pads with the median value of all or part of the vector along each axis.

‘minimum’ Pads with the minimum value of all or part of the vector along each axis.

‘reflect’ Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.

‘symmetric’ Pads with the reflection of the vector mirrored along the edge of the array.

‘wrap’ Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

<function> Padding function, see Notes.

**stat_length** [sequence or int, optional] Used in ‘maximum’, ‘mean’, ‘median’, and ‘minimum’. Number of values at edge of each axis used to calculate the statistic value.

((before_1, after_1), ... (before_N, after_N)) unique statistic lengths for each axis.

((before, after),) yields same before and after statistic lengths for each axis.

(stat_length,) or int is a shortcut for before = after = statistic length for all axes.

Default is `None`, to use the entire axis.

**constant_values** [sequence or int, optional] Used in ‘constant’. The values to set the padded values for each axis.

((before_1, after_1), ... (before_N, after_N)) unique pad constants for each axis.

((before, after),) yields same before and after constants for each axis.

(constant,) or int is a shortcut for before = after = constant for all axes.

Default is 0.

**end_values** [sequence or int, optional] Used in ‘linear_ramp’. The values used for the ending value of the linear_ramp and that will form the edge of the padded array.

((before_1, after_1), ... (before_N, after_N)) unique end values for each axis.

((before, after),) yields same before and after end values for each axis.

(constant,) or int is a shortcut for before = after = end value for all axes.

Default is 0.

**reflect_type** [{'even', 'odd'}, optional] Used in ‘reflect’, and ‘symmetric’. The ‘even’ style is the default with an unaltered reflection around the edge value. For the ‘odd’ style, the extended part of the array is created by subtracting the reflected values from two times the edge value.

**Returns**

**pad** [ndarray] Padded array of rank equal to `array` with shape increased according to `pad_width`. 
Notes

New in version 1.7.0.

For an array with rank greater than 1, some of the padding of later axes is calculated from padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded array are calculated by using padded values from the first axis.

The padding function, if used, should return a rank 1 array equal in length to the vector argument with padded values replaced. It has the following signature:

```
padding_func(vector, iaxis_pad_width, iaxis, kwargs)
```

where

- **vector** [ndarray] A rank 1 array already padded with zeros. Padded values are vector[:pad_tuple[0]] and vector[-pad_tuple[1]:].
- **iaxis_pad_width** [tuple] A 2-tuple of ints, iaxis_pad_width[0] represents the number of values padded at the beginning of vector where iaxis_pad_width[1] represents the number of values padded at the end of vector.
- **iaxis** [int] The axis currently being calculated.
- **kwargs** [dict] Any keyword arguments the function requires.

Examples

```python
>>> a = [1, 2, 3, 4, 5]
>>> np.pad(a, (2,3), 'constant', constant_values=(4, 6))
array([4, 4, 1, 2, 3, 4, 5, 6, 6, 6])

>>> np.pad(a, (2, 3), 'edge')
array([1, 1, 1, 2, 3, 4, 5, 5, 5, 5])

>>> np.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([5, 3, 3, 2, 1, 2, 3, 4, 5, 2, -1, -4])

>>> np.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])

>>> np.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])

>>> np.pad(a, (2,), 'median')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])

>>> a = [[1, 2], [3, 4]]
>>> np.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
[1, 1, 1, 2, 1, 1, 1],
[1, 1, 1, 2, 1, 1, 1],
[3, 3, 3, 4, 3, 3, 3],
[1, 1, 1, 2, 1, 1, 1],
[1, 1, 1, 2, 1, 1, 1]])
```
```python
>>> a = [1, 2, 3, 4, 5]
>>> np.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])

>>> np.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1, 0, 1, 2, 3, 4, 5, 6, 7, 8])

>>> np.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])

>>> np.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])

>>> np.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])

>>> def pad_with(vector, pad_width, iaxis, kwargs):
...     pad_value = kwargs.get('padder', 10)
...     vector[:pad_width[0]] = pad_value
...     vector[-pad_width[1]:] = pad_value
...     return vector

>>> a = np.arange(6)
>>> a = a.reshape((2, 3))

>>> np.pad(a, 2, pad_with)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 0, 1, 2, 10, 10],
       [10, 10, 3, 4, 5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])

>>> np.pad(a, 2, pad_with, padder=100)
array([[100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 0, 1, 2, 100, 100],
       [100, 100, 3, 4, 5, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100]])
```

dask.array.percentile(a, q, interpolation='linear', method='default')

Approximate percentile of 1-D array

Parameters

**a** [Array]

**q** [array_like of float] Percentile or sequence of percentiles to compute, which must be between 0 and 100 inclusive.

**interpolation** [{‘linear’, ‘lower’, ‘higher’, ‘midpoint’, ‘nearest’}, optional] The interpolation method to use when the desired percentile lies between two data points \(i < j\). Only valid for method=’dask’.

- ‘linear’: \(i + (j - i) \times \text{fraction}\), where \(\text{fraction}\) is the fractional part of the index surrounded by \(i\) and \(j\).
- ‘lower’: \(i\).
- ‘higher’: \(j\).
- ‘nearest’: \(i\) or \(j\), whichever is nearest.
- ‘midpoint’: \((i + j) / 2\).
method  [['default', 'dask', 'tdigest'], optional] What method to use. By default will use dask’s internal custom algorithm ('dask'). If set to 'tdigest' will use tdigest for floats and ints and fallback to the 'dask' otherwise.

See also:

numpy.percentile  Numpy’s equivalent Percentile function

dask.array.piecewise(x, condlist, funclist, *args, **kw)
Evaluate a piecewise-defined function.

Given a set of conditions and corresponding functions, evaluate each function on the input data wherever its condition is true.

Parameters

x [ndarray or scalar] The input domain.

condlist [list of bool arrays or bool scalars] Each boolean array corresponds to a function in funclist. Wherever condlist[i] is True, funclist[i](x) is used as the output value.

Each boolean array in condlist selects a piece of x, and should therefore be of the same shape as x.

The length of condlist must correspond to that of funclist. If one extra function is given, i.e.
if len(funclist) == len(condlist) + 1, then that extra function is the default value, used wherever all conditions are false.

funclist [list of callables, f(x,*args,**kw), or scalars] Each function is evaluated over x wherever its corresponding condition is True. It should take a 1d array as input and give an 1d array or a scalar value as output. If, instead of a callable, a scalar is provided then a constant function (lambda x: scalar) is assumed.

args [tuple, optional] Any further arguments given to piecewise are passed to the functions upon execution, i.e., if called piecewise(..., ..., 1, 'a'), then each function is called as f(x, 1, 'a').

kw [dict, optional] Keyword arguments used in calling piecewise are passed to the functions upon execution, i.e., if called piecewise(..., ..., alpha=1), then each function is called as f(x, alpha=1).

Returns

out [ndarray] The output is the same shape and type as x and is found by calling the functions in funclist on the appropriate portions of x, as defined by the boolean arrays in condlist. Portions not covered by any condition have a default value of 0.

See also:

choose, select, where

Notes

This is similar to choose or select, except that functions are evaluated on elements of x that satisfy the corresponding condition from condlist.

The result is:
Examples

Define the sigma function, which is \(-1\) for \(x < 0\) and \(+1\) for \(x \geq 0\).

```python
>>> x = np.linspace(-2.5, 2.5, 6)
>>> np.piecewise(x, [x < 0, x >= 0], [-1, 1])
array([-1., -1., -1., 1., 1., 1.])
```

Define the absolute value, which is \(-x\) for \(x < 0\) and \(x\) for \(x \geq 0\).

```python
>>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
array([2.5, 1.5, 0.5, 0.5, 1.5, 2.5])
```

Apply the same function to a scalar value.

```python
>>> y = -2
>>> np.piecewise(y, [y < 0, y >= 0], [lambda x: -x, lambda x: x])
array(2)
```

dask.array.prod(a, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>)

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 be broadcast correctly against the input array.

  If the default value is passed, then keepdims will not be passed through to the prod method of sub-classes of ndarray, however any non-default value will be. If the sub-class’ method does not implement keepdims any exceptions will be raised.
initial [scalar, optional] The starting value for this product. See ~numpy.ufunc.reduce for details.

New in version 1.15.0.

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:
You can also start the product with a value other than one:

```python
>>> np.prod([1, 2], initial=5)
10
```

dask.array.ptp

```
dask.array.ptp (a, axis=None, out=None, keepdims=<no value>)
```

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** [None or int or tuple of ints, optional] Axis along which to find the peaks. By default, flatten the array. axis may be negative, in which case it counts from the last to the first axis.
  
  New in version 1.15.0.

  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** [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.

- **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.

  If the default value is passed, then keepdims will not be passed through to the ptp method of sub-classes of ndarray, however any non-default value will be. If the sub-class’ method does not implement keepdims any exceptions will be raised.

**Returns**

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

**Examples**

```python
>>> 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

```
dask.array.rad2deg (x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])
```

Convert angles from radians to degrees.
Parameters

- **x** [array_like] Angle in radians.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [ndarray] The corresponding angle in degrees. This is a scalar if x is a scalar.

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

```python
>>> np.rad2deg(np.pi/2)  # doctest: +SKIP
90.0
```

dask.array.radians(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Convert angles from degrees to radians.

Parameters

- **x** [array_like] Input array in degrees.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [ndarray] The corresponding radian values. This is a scalar if x is a scalar.

See also:

- **deg2rad** equivalent function
Examples

Convert a degree array to radians

```python
>>> deg = np.arange(12.) * 30.  # doctest: +SKIP
>>> np.radians(deg)  # doctest: +SKIP
array([ 0.      , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265, 3.66519143, 4.1887902 , 4.71238898,
       5.23598776, 5.75958653])
```

```python
>>> out = np.zeros((deg.shape))  # doctest: +SKIP
>>> ret = np.radians(deg, out)  # doctest: +SKIP
>>> ret is out  # doctest: +SKIP
True
```

*dask.array.ravel*(\(a, \texttt{order}='C'\))

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 \texttt{order}, and packed as a 1-D array.

\texttt{order} [[\texttt{C}'',\texttt{F}'', \texttt{A}'', \texttt{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 contiguos 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] \(y\) is an array of the same subtype as \(a\), with shape \((a.size,)\). Note that matrices are special cased for backward compatibility, if \(a\) is a matrix, then \(y\) is a 1-D ndarray.

**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)`.

```python
g = np.array([[1, 2, 3], [4, 5, 6]])
print(g.ravel())
```

```python
print(g.reshape(-1))
```

```python
print(g.ravel(order='F'))
```

When `order` is ‘A’, it will preserve the array’s ‘C’ or ‘F’ ordering:

```python
print(g.T.ravel())
```

```python
print(g.T.ravel(order='A'))
```

When `order` is ‘K’, it will preserve orderings that are neither ‘C’ nor ‘F’, but won’t reverse axes:

```python
a = np.arange(3)[::-1]; a
a.ravel(order='C')
a.ravel(order='K')
```

```python
da = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
```

```python
da.ravel(order='C')
da.ravel(order='K')
```

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

Return the real part of the complex argument.

**Parameters**

- `val` [array_like] Input array.

**Returns**

- `out` [ndarray or scalar] The real component of the complex argument. If `val` is real, the type of `val` is used for the output. If `val` has complex elements, the returned type is float.

**See also:**

- `real_if_close`, `imag`, `angle`
Examples

```python
>>> a = np.array([1+2j, 3+4j, 5+6j])  # doctest: +SKIP
>>> a.real  # doctest: +SKIP
darray([1., 3., 5.])
>>> a.real = 9  # doctest: +SKIP
>>> a  # doctest: +SKIP
darray([9.+2.j, 9.+4.j, 9.+6.j])
>>> a.real = np.array([9, 8, 7])  # doctest: +SKIP
>>> a  # doctest: +SKIP
darray([9.+2.j, 8.+4.j, 7.+6.j])
>>> np.real(1 + 1j)  # doctest: +SKIP
1.0
```

dask.array.rechunk(x, chunks, threshold=None, block_size_limit=None)
Convert blocks in dask array x for new chunks.

Parameters

- **x**: dask array  Array to be rechunked.
- **chunks**: int, tuple or dict  The new block dimensions to create. -1 indicates the full size of the corresponding dimension.
- **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 Defaults to the configuration value array.chunk-size

Examples

```python
>>> import dask.array as da
>>> x = da.ones((1000, 1000), chunks=(100, 100))
Specify uniform chunk sizes with a tuple
>>> y = x.rechunk((1000, 10))
Or chunk only specific dimensions with a dictionary
>>> y = x.rechunk({0: 1000})
Use the value -1 to specify that you want a single chunk along a dimension or the value "auto" to specify that dask can freely rechunk a dimension to attain blocks of a uniform block size
>>> y = x.rechunk({0: -1, 1: 'auto'}, block_size_limit=1e8)
```

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

```python
>>> np.repeat(3, 4)
array([3, 3, 3, 3])
>>> x = np.array([[1,2],[3,4]])
>>> np.repeat(x, 2)
array([[1, 1, 2, 2, 3, 3, 4, 4]], dtype=int32)
>>> np.repeat(x, 3, axis=1)
array([[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]], dtype=int32)
>>> 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 'row-major order'_.
2. It only allows for reshapings that collapse or merge dimensions like (1, 2, 3, 4) -> (1, 6, 4) or (64,) -> (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

```python
>>> 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=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Round elements of the array to the nearest integer.

Parameters

x [array_like] Input array.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

out [ndarray or scalar] Output array is same shape and type as x. This is a scalar if x is a scalar.
See also:

`ceil`, `floor`, `trunc`

**Examples**

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])  # doctest: +SKIP
>>> np.rint(a)  # doctest: +SKIP
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 or tuple of ints] The number of places by which elements are shifted. If a tuple, then `axis` must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If an int while `axis` is a tuple of ints, then the same value is used for all given axes.
- `axis` [int or tuple of ints, optional] Axis or axes 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.

**Notes**

New in version 1.12.0.

Supports rolling over multiple dimensions simultaneously.

**Examples**

```python
>>> 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))
narray([[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]])
```
dask Documentation, Release 1.2.2

```python
>>> 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.

**See also:**

around equivalent function; see for details.

dask.array.sign(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True)

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 sign(x.real) + 0j if x.real != 0 else sign(x.imag) + 0j.

complex(nan, 0) is returned for complex nan inputs.

**Parameters**

- x [array_like] Input values.
- out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**Returns**

- y [ndarray] The sign of x. This is a scalar if x is a scalar.

**Notes**

There is more than one definition of sign in common use for complex numbers. The definition used here is equivalent to \( x / \sqrt{x \times \overline{x}} \) which is different from a common alternative, \( x / |x| \).

**Examples**

```python
>>> np.sign([-5., 4.5])  # doctest: +SKIP
array([-1., 1.])
>>> np.sign(0)           # doctest: +SKIP
0
>>> np.sign(5-2j)        # doctest: +SKIP
(1+0j)
```

dask.array.signbit(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True)

Returns element-wise True where signbit is set (less than zero).
Parameters

x [array_like] The input value(s).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

result [ndarray of bool] Output array, or reference to `out` if that was supplied. This is a scalar if `x` is a scalar.

Examples

```python
>>> np.signbit(-1.2)  # doctest: +SKIP
True
>>> np.signbit(np.array([1, -2.3, 2.1]))  # doctest: +SKIP
array([False, True, False])
```

dask.array.sin(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Trigonometric sine, element-wise.

Parameters

x [array_like] Angle, in radians (2π rad equals 360 degrees).

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [array_like] The sine of each element of `x`. This is a scalar if `x` is a scalar.

See also:

arcsin, sinh, cos

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the +x axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray’s intersection with the unit circle is the sine of that angle. It ranges from -1 for x = 3π/2 to +1 for π/2. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.
Examples

Print sine of one angle:

```python
>>> np.sin(np.pi/2.)  # doctest: +SKIP
1.0
```

Print sines of an array of angles given in degrees:

```python
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )  # doctest: +SKIP
array([ 0. , 0.5 , 0.70710678, 0.8660254 , 1. ])
```

Plot the sine function:

```python
>>> import matplotlib.pylab as plt  # doctest: +SKIP
>>> x = np.linspace(-np.pi, np.pi, 201)  # doctest: +SKIP
>>> plt.plot(x, np.sin(x))  # doctest: +SKIP
>>> plt.xlabel('Angle [rad]')  # doctest: +SKIP
>>> plt.ylabel('sin(x)')  # doctest: +SKIP
>>> plt.axis('tight')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

dask.array.sinh(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Hyperbolic sine, element-wise.

Equivalent to \(1/2 * (\exp(x) - \exp(-x))\) or \(-1j * \sin(1j * x)\).

Parameters

- **x** [array_like] Input array.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs
  - **casting** For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [ndarray] The corresponding hyperbolic sine values. This is a scalar if x is a scalar.

Notes

If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)

References

Examples

```python
>>> np.sinh(0)  # doctest: +SKIP
0.0
>>> np.sinh(np.pi*1j/2)  # doctest: +SKIP
1j
>>> np.sinh(np.pi*1j)  # (exact value is 0)  # doctest: +SKIP
1.2246063538223773e-016j
>>>  # Discrepancy due to vagaries of floating point arithmetic.
```

```python
>>> # Example of providing the optional output parameter
>>> out2 = np.sinh([0.1], out1)  # doctest: +SKIP
>>> out2 is out1  # doctest: +SKIP
True
```

```python
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.sinh(np.zeros((3,3)),np.zeros((2,2)))  # doctest: +SKIP
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

dask.array.sqrt(x, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

Return the non-negative square-root of an array, element-wise.

Parameters

x [array_like] The values whose square-roots are required.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray] An array of the same shape as x, containing the positive square-root of each element in x. If any element in x is complex, a complex array is returned (and the square-roots of negative reals are calculated). If all of the elements in x are real, so is y, with negative elements returning nan. If out was provided, y is a reference to it. This is a scalar if x is a scalar.

See also:

lib.scimath.sqrt A version which returns complex numbers when given negative reals.

Notes

sqrt has–consistent with common convention–as its branch cut the real “interval” [-inf, 0), and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.
Examples

```python
>>> np.sqrt([1,4,9])  # doctest: +SKIP
array([ 1.,  2.,  3.])

>>> np.sqrt([4, -1, -3+4J])  # doctest: +SKIP
array([ 2.+0.j,  0.+1.j,  1.+2.j])

>>> np.sqrt([4, -1, numpy.inf])  # doctest: +SKIP
array([ 2., NaN, Inf])
```

dask.array.square(x, /, out=None, *, where=True, casting='same_kind', order='K', subok=True, signature=None, extobj=None)

Return the element-wise square of the input.

Parameters

- `x` [array_like] Input data.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- `out` [ndarray or scalar] Element-wise \(x^2\), of the same shape and dtype as \(x\). This is a scalar if \(x\) is a scalar.

See also:

- `numpy.linalg.matrix_power`, `sqrt`, `power`

Examples

```python
>>> np.square([-1j, 1])  # doctest: +SKIP
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`.

 Raises
**ValueError**  If axis is not None, and an axis being squeezed is not of length 1

See also:

- **expand_dims**  The inverse operation, adding singleton dimensions
- **reshape**  Insert, remove, and combine dimensions, and resize existing ones

### Examples

```python
>>> x = np.array([[0], [1], [2]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=0).shape
(3, 1)
>>> np.squeeze(x, axis=1).shape
Traceback (most recent call last):
  ... 
ValueError: cannot select an axis to squeeze out which has size not equal to one
>>> 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

```python
>>> 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=<no value>)

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.
  
  New in version 1.7.0.

  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 \( N - ddof \), 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 input array.

  If the default value is passed, then `keepdims` will not be passed through to the `std` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class’ method does not implement `keepdims` any exceptions will be raised.

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., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - \text{mean}(x))^2)} \).

The average squared deviation is normally calculated as \( x \cdot \text{sum}() / N \), where \( N = \text{len}(x) \). 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

```python
g = np.array([[1, 2], [3, 4]])
np.std(g)  # Default float32 precision
1.1180339887498949
np.std(g, axis=0)  # Axis 0
array([1., 1.])
np.std(g, axis=1)  # Axis 1
array([0.5, 0.5])
```

In single precision, std() can be inaccurate:

```python
g = np.zeros((2, 512*512), dtype=np.float32)
g[0, :] = 1.0
g[1, :] = 0.1
np.std(g)  # Default float32 precision
0.45000005
```

Computing the standard deviation in float64 is more accurate:

```python
g = np.zeros((2, 512*512), dtype=np.float64)
g[0, :] = 1.0
g[1, :] = 0.1
np.std(g)  # Default float64 precision
0.44999999925494177
```

dask.array.sum(a, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>)

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 dtypes 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.

  If the default value is passed, then keepdims will not be passed through to the sum method of sub-classes of ndarray, however any non-default value will be. If the sub-class’ method does not implement keepdims any exceptions will be raised.
**initial** [scalar, optional] Starting value for the sum. See ~numpy.ufunc.reduce for details.

New in version 1.15.0.

**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:

```python
>>> np.sum([])
0.0
```

**Examples**

```python
>>> 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:

```python
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

You can also start the sum with a value other than zero:

```python
>>> np.sum([10], initial=5)
15
```

*dask.array.take* *(a, indices, axis=None, out=None, mode='raise')*

Take elements from an array along an axis.

When axis is not None, 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. A call such as `np.take(arr, indices, axis=3)` is equivalent to `arr[:,:,:,:,indices,...]`. 
Explained without fancy indexing, this is equivalent to the following use of \texttt{ndindex}, which sets each of $ii$, $jj$, and $kk$ to a tuple of indices:

\begin{verbatim}
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
Nj = indices.shape
for ii in ndindex(Ni):
    for jj in ndindex(Nj):
        for kk in ndindex(Nk):
            out[ii + jj + kk] = a[ii + (indices[jj],) + kk]
\end{verbatim}

\begin{description}
\item[Parameters]
\begin{itemize}
\item \texttt{a} [array_like (Ni, $\ldots$, M, Nk, $\ldots$)] The source array.
\item \texttt{indices} [array_like (Nj, $\ldots$)] The indices of the values to extract.
\end{itemize}
\begin{itemize}
\item New in version 1.8.0. Also allow scalars for indices.
\item \texttt{axis} [int, optional] The axis over which to select values. By default, the flattened input array is used.
\item \texttt{out} [ndarray, optional (Ni, $\ldots$, Nj, $\ldots$, Nk, $\ldots$)] If provided, the result will be placed in this array. It should be of the appropriate shape and dtype.
\item \texttt{mode} [{‘raise’, ‘wrap’, ‘clip’}, optional] Specifies how out-of-bounds indices will behave.
\begin{itemize}
\item ‘raise’ – raise an error (default)
\item ‘wrap’ – wrap around
\item ‘clip’ – clip to the range
\end{itemize}
\end{itemize}
\item[Returns]
\begin{itemize}
\item \texttt{out} [ndarray (Ni, $\ldots$, Nj, $\ldots$, Nk, $\ldots$)] The returned array has the same type as \texttt{a}.
\end{itemize}
\end{description}

\begin{description}
\item[See also:]
\begin{itemize}
\item \texttt{compress} Take elements using a boolean mask
\item \texttt{ndarray.take} equivalent method
\item \texttt{take_along_axis} Take elements by matching the array and the index arrays
\end{itemize}
\end{description}

\begin{description}
\item[Notes]
By eliminating the inner loop in the description above, and using $s_\_\_\_\_$ to build simple slice objects, \texttt{take} can be expressed in terms of applying fancy indexing to each 1-d slice:
\begin{verbatim}
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        out[ii + s_[...], + kk] = a[ii + s_[...], + kk][indices]
\end{verbatim}
\end{description}

For this reason, it is equivalent to (but faster than) the following use of \texttt{apply_along_axis}:
```python
out = np.apply_along_axis(lambda a_1d: a_1d[indices], axis, a)
```

### Examples

```python
>>> 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.

```python
>>> a = np.array(a)
>>> a[indices]
array([4, 3, 6])
```

If `indices` is not one dimensional, the output also has these dimensions.

```python
>>> np.take(a, [[0, 1], [2, 3]])
array([[4, 3],
       [5, 7]])
```

dask.array.tan(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

Compute tangent element-wise.

Equivalent to `np.sin(x)/np.cos(x)` element-wise.

**Parameters**

- `x` [array_like] Input array.
- `out` [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- `where` [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- **`**kwargs** For other keyword-only arguments, see the ufunc docs.

**Returns**

- `y` [ndarray] The corresponding tangent values. This is a scalar if `x` is a scalar.

**Notes**

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

### References

Examples

```python
>>> from math import pi  # doctest: +SKIP
>>> np.tan(np.array([-pi, pi/2, pi]))  # doctest: +SKIP
array([ 1.22460635e-16, 1.63317787e+16, -1.22460635e-16])
```

```python
# Example of providing the optional output parameter illustrating
# that what is returned is a reference to said parameter
>>> out2 = np.cos([0.1], out1)  # doctest: +SKIP
>>> out2 is out1  # doctest: +SKIP
True
```

```python
# Example of ValueError due to provision of shape mis-matched 'out'
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))  # doctest: +SKIP
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

dask.array.tanh(x, out=None, *, where=True, casting=’same_kind’, order=’K’, subok=True, signature=None, extobj=None)
Compute hyperbolic tangent element-wise.

Equivalent to np.sinh(x)/np.cosh(x) or -1j * np.tan(1j*x).

Parameters

- **x** [array_like] Input array.
- **out** [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.
- ****kwargs For other keyword-only arguments, see the ufunc docs.

Returns

- **y** [ndarray] The corresponding hyperbolic tangent values. This is a scalar if x is a scalar.

Notes

If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)

References

[1], [2]

Examples

```python
>>> np.tanh((0, np.pi*1j, np.pi*1j/2))  # doctest: +SKIP
array([ 0. +0.00000000e+00j, 0. -1.22460635e-16j, 0. +1.63317787e+16j])
```
```python
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out2 = np.tanh([0.1], out1)  # doctest: +SKIP
>>> out2 is out1  # doctest: +SKIP
True
```

dask.array.tensordot(a, b, axes=2)

Compute tensor dot product along specified axes for arrays >= 1-D.

Given two tensors (arrays of dimension greater than or equal to one), `a` and `b`, and an array_like object containing two array_like objects, `(a_axes, b_axes)`, sum the products of `a`'s and `b`'s elements (components) over the axes specified by `a_axes` and `b_axes`. The third argument can be a single non-negative integer_like scalar, `N`; if it is such, then the last `N` dimensions of `a` and the first `N` dimensions of `b` are summed over.

**Parameters**

- `a, b` [array_like, len(shape) >= 1] Tensors to “dot”.
- `axes` [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 \( a \otimes b \)
- `axes = 1`: tensor dot product \( a \cdot b \)
- `axes = 2` (default) tensor double contraction \( a : b \)

When `axes` is integer_like, the sequence for evaluation will be: first the -Nth axis in `a` and 0th axis in `b`, and the -1th axis in `a` and Nth 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.,  5162.],
        [ 4796.,  5306.],
        [ 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]])

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([abbcccdddd, aaaaabbbbbccccccccdddddddd], dtype=object)

>>> np.tensordot(a, A, 1)
array([[aaacccc, bbbddddd],
       [[aaaaaccccc, bbbbbdddddd],
       [aaaaaaaccccccc, bbbbbbbddddddddd]], 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, cdddddd],
        [aabbbbbb, cccdddddd]],
        ...

(continues on next page)
\[
[[aaabbbbbbb, cccddddddd],
 [aaabbbbbbb, cccddddddd]],
\text{dtype=object})
\]

```python
>>> np.tensordot(a, A, (2, 1))
array([[abb, cdd],
       [aaabbbb, ccccdddd]),
      [aaaaabbbbbbb, cccccdddddddd]],
     dtype=object)
```

```python
>>> np.tensordot(a, A, ((0, 1), (0, 1)))
array([abbbccccddddddd, aabbbbccccccdddddddd],
      dtype=object)
```

```python
>>> np.tensordot(a, A, ((2, 1), (1, 0)))
array([acccbbdddd, aaaaacccccccbbbbbbddddd],
      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.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.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**

```python
>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])
```

```python
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
```

```python
>>> np.tile(a, (2, 1, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
```
```python
>>> 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(a, k, axis=-1, split_every=None)

Extract the k largest elements from a on the given axis, and return them sorted from largest to smallest. If k is negative, extract the -k smallest elements instead, and return them sorted from smallest to largest.

This performs best when k is much smaller than the chunk size. All results will be returned in a single chunk along the given axis.

**Parameters**

- **x**: Array  Data being sorted
- **k**: int
- **axis**: int, optional
- **split_every**: int >=2, optional  See reduce(). This parameter becomes very important when k is on the same order of magnitude of the chunk size or more, as it prevents getting the whole or a significant portion of the input array in memory all at once, with a negative impact on network transfer too when running on distributed.

**Returns**

Selection of x with size abs(k) along the given axis.

**Examples**

```python
>>> import dask.array as da
>>> x = np.array([5, 1, 3, 6])
>>> d = da.from_array(x, chunks=2)
>>> d.topk(2).compute()
array([6, 5])
>>> d.topk(-2).compute()
array([1, 3])
```

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

```python
>>> 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=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True, signature=None, extobj=None)

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.

out [ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs For other keyword-only arguments, see the ufunc docs.

Returns

y [ndarray or scalar] The truncated value of each element in x. This is a scalar if x is a scalar.

See also:

ceil, floor, rint

Notes

New in version 1.3.0.

Examples

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])  # doctest: +SKIP
>>> np.trunc(a)  # doctest: +SKIP
array([-1., -1., -0., 0., 1., 1., 2.])
```

dask.array.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None)

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
- the number of times each unique value comes up in the input array

Parameters
ar  [array_like] Input array. Unless axis is specified, this will be flattened if it is not already 1-D.

return_index  [bool, optional] If True, also return the indices of ar (along the specified axis, if provided, or in the flattened array) that result in the unique array.

return_inverse  [bool, optional] If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct ar.

return_counts  [bool, optional] If True, also return the number of times each unique item appears in ar.

New in version 1.9.0.

axis  [int or None, optional] The axis to operate on. If None, ar will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis, see the notes for more details. Object arrays or structured arrays that contain objects are not supported if the axis kwarg is used. The default is None.

New in version 1.13.0.

Returns

unique  [ndarray] The sorted unique values.

unique_indices  [ndarray, optional] The indices of the first occurrences of the unique values in the original array. Only provided if return_index is True.

unique_inverse  [ndarray, optional] The indices to reconstruct the 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.

Notes

When an axis is specified the subarrays indexed by the axis are sorted. This is done by making the specified axis the first dimension of the array and then flattening the subarrays in C order. The flattened subarrays are then viewed as a structured type with each element given a label, with the effect that we end up with a 1-D array of structured types that can be treated in the same way as any other 1-D array. The result is that the flattened subarrays are sorted in lexicographic order starting with the first element.

Examples

```python
>>> 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 unique rows of a 2D array
>>> a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
>>> np.unique(a, axis=0)
array([[1, 0, 0], [2, 3, 4]])

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)

>>> indices
array([0, 1, 3])

>>> a[indices]
aarray(['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]
aarray([1, 2, 6, 4, 2, 3, 2])

dask.array.unravel_index(indices, shape, order='C')
Converts a flat index or array of flat indices into a tuple of coordinate arrays.

**Parameters**

- **indices** [array_like] An integer array whose elements are indices into the flattened version of
  an array of dimensions shape. Before version 1.6.0, this function accepted just one index value.

- **shape** [tuple of ints] The shape of the array to use for unraveling indices.

  Changed in version 1.16.0: Renamed from dims to shape.

- **order** ['C', 'F'], optional Determines whether the indices should be viewed as indexing in
  row-major (C-style) or column-major (Fortran-style) order.

  New in version 1.6.0.

**Returns**

- **unraveled_coords** [tuple of ndarray] Each array in the tuple has the same shape as the
  indices array.

**See also:**

ravel_multi_index

**Examples**
dask documentation, release 1.2.2

```python
>>> np.unravel_index([22, 41, 37], (7,6))
(array([3, 6, 6]), array([4, 5, 1]))
```
```
>>> np.unravel_index([31, 41, 13], (7,6), order='F')
(array([3, 6, 6]), array([4, 5, 1]))
```
```
>>> np.unravel_index(1621, (6,7,8,9))
(3, 1, 4, 1)
```

dask.array.var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>)

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.
  - New in version 1.7.0.
  - 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 \( N - ddof \), 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 input array.
  - If the default value is passed, then `keepdims` will not be passed through to the `var` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class’ method does not implement `keepdims` any exceptions will be raised.

**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 \( \frac{x.\text{sum}}{N} \), where \( N = \text{len}(x) \). 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 \texttt{float32} (see example below). Specifying a higher-accuracy accumulator using the \texttt{dtype} keyword can alleviate this issue.

### Examples

```python
>>> 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, \texttt{var()} can be inaccurate:

```python
>>> 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 \texttt{float64} is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932944759
```

```
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.2025
```

dask\texttt{.array.vdot}(a, b)

Return the dot product of two vectors.

The \texttt{vdot}(a, b) function handles complex numbers differently than \texttt{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 \texttt{vdot} handles multidimensional arrays differently than \texttt{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

- \( \text{output} \) [ndarray] Dot product of \( a \) and \( b \). Can be an int, float, or complex depending on the types of \( a \) and \( b \).

See also:

4.7. Array
**dot**  Return the dot product without using the complex conjugate of the first argument.

**Examples**

```python
>>> 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!

```python
>>> 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.**vstack**(tup)

Stack arrays in sequence vertically (row wise).

This is equivalent to concatenation along the first axis after 1-D arrays of shape \((N,)\) have been reshaped to \((I, N)\). Rebuilds arrays divided by **vsplit**.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions **concatenate**, **stack** and **block** provide more general stacking and concatenation operations.

**Parameters**

- **tup** [sequence of ndarrays] The arrays must have the same shape along all but the first axis. 1-D arrays must have the same length.

**Returns**

- **stacked** [ndarray] The array formed by stacking the given arrays, will be at least 2-D.

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.
- **block**  Assemble arrays from blocks.

**Examples**
```python
>>> 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 chosen from x or y depending on condition.

**Note:** When only condition is provided, this function is a shorthand for np.asarray(condition). nonzero(). Using nonzero directly should be preferred, as it behaves correctly for subclasses. The rest of this documentation covers only the case where all three arguments are provided.

**Parameters**

- condition [array_like, bool] Where True, yield x, otherwise yield y.
- x, y [array_like] Values from which to choose. x, y and condition need to be broadcastable to some shape.

**Returns**

- out [ndarray] An array with elements from x where condition is True, and elements from y elsewhere.

**See also:**

choose

nonzero The function that is called when x and y are omitted

**Notes**

If all the arrays are 1-D, where is equivalent to:

```python
[xv if c else yv
 for c, xv, yv in zip(condition, x, y)]
```

**Examples**

```python
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```
>>> np.where(a < 5, a, 10*a)
array([ 0, 1, 2, 3, 4, 50, 60, 70, 80, 90])

This can be used on multidimensional arrays too:

```python
>>> np.where(([True, False], [True, True]),
          [[1, 2], [3, 4]],
          [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])
```

The shapes of x, y, and the condition are broadcast together:

```python
>>> x, y = np.ogrid[:3, :4]
>>> np.where(x < y, x, 10 + y)  # both x and 10+y are broadcast
array([[10, 0, 0, 0],
       [10, 11, 1, 1],
       [10, 11, 12, 2]])
```

```python
>>> a = np.array([[0, 1, 2],
                [0, 2, 4],
                [0, 3, 6]])
>>> np.where(a < 4, a, -1)  # -1 is broadcast
array([[ 0, 1, 2],
       [ 0, 2, -1],
       [ 0, 3, -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 tuple 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, default: ‘C’] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) 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.
- **empty** Return a new uninitialized array.
- **ones** Return a new array setting values to one.
- **full** Return a new array of given shape filled with value.
Examples

```python
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])

>>> np.zeros((5,), dtype=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**


4.7. Array
dask.array.linalg.inv(a)
Compute the inverse of a matrix with LU decomposition and forward / backward substitutions.

Parameters
a [array_like] Square matrix to be inverted.

Returns
ainv [Array] Inverse of the matrix a.

dask.array.linalg.lstsq(a, b)
Return the least-squares solution to a linear matrix equation using QR decomposition.

Solves the equation $a x = b$ by computing a vector $x$ that minimizes the Euclidean 2-norm $\| b - a x \|^2$. The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of $a$ can be less than, equal to, or greater than its number of linearly independent columns). If $a$ is square and of full rank, then $x$ (but for round-off error) is the “exact” solution of the equation.

Parameters
a [(M, N) array_like] “Coefficient” matrix.
b [(M,) array_like] Ordinate or “dependent variable” values.

Returns
x [(N,) Array] Least-squares solution. If $b$ is two-dimensional, the solutions are in the $K$ columns of $x$.
residuals [(1,) Array] Sums of residuals; squared Euclidean 2-norm for each column in $b - a*x$.
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)  # doctest: +SKIP
```

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.
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.

New in version 1.8.0.

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

\[ n \quad \text{[float or ndarray]} \quad \text{Norm of the matrix or vector(s)}. \]

Notes

For values of \( \text{ord} <= 0 \), 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:

<table>
<thead>
<tr>
<th>ord</th>
<th>norm for matrices</th>
<th>norm for vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Frobenius norm</td>
<td>2-norm</td>
</tr>
<tr>
<td>‘fro’</td>
<td>Frobenius norm</td>
<td>–</td>
</tr>
<tr>
<td>‘nuc’</td>
<td>nuclear norm</td>
<td>–</td>
</tr>
<tr>
<td>inf</td>
<td>( \max(\text{sum(abs}(x), \text{axis}=1)) )</td>
<td>( \max(\text{abs}(x)) )</td>
</tr>
<tr>
<td>-inf</td>
<td>( \min(\text{sum(abs}(x), \text{axis}=1)) )</td>
<td>( \min(\text{abs}(x)) )</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>( \text{sum}(x \neq 0) )</td>
</tr>
<tr>
<td>1</td>
<td>( \max(\text{sum(abs}(x), \text{axis}=0)) )</td>
<td>as below</td>
</tr>
<tr>
<td>-1</td>
<td>( \min(\text{sum(abs}(x), \text{axis}=0)) )</td>
<td>as below</td>
</tr>
<tr>
<td>2</td>
<td>2-norm (largest sing. value)</td>
<td>as below</td>
</tr>
<tr>
<td>-2</td>
<td>smallest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>other</td>
<td>–</td>
<td>( \text{sum}(\text{abs}(x)<strong>\text{ord})</strong>(1./\text{ord}) )</td>
</tr>
</tbody>
</table>

The Frobenius norm is given by [1]:

\[ ||A||_F = \left( \sum_{i,j} \text{abs}(a_{i,j})^2 \right)^{1/2} \]

The nuclear norm is the sum of the singular values.

References

[1]

Examples

```python
>>> 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))
(continues on next page)
```
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,  2.3606798,   5.
       ])
>>> LA.norm(c, axis=1)
array([ 3.74165739,  4.24264069])
```

Using the `axis` argument to compute matrix norms:
```python
>>> 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.<code>qr</code>(a)

Compute the qr factorization of a matrix.

**Returns**

- **q**: Array, orthonormal
- **r**: Array, upper-triangular

**See also:**

- <code>np.linalg.qr</code> Equivalent NumPy Operation
- <code>dask.array.linalg.tsqr</code> Implementation for tall-and-skinny arrays
- <code>dask.array.linalg.sfqr</code> Implementation for short-and-fat arrays

**Examples**

```python
>>> q, r = da.linalg.qr(x)  # doctest: +SKIP
```

dask.array.linalg.<code>solve</code>(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.<code>solve_triangular</code>(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.<code>svd</code>(a)

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` Implementation for tall-and-skinny arrays

Examples

```python
>>> u, s, v = da.linalg.svd(x)  # doctest: +SKIP
```

**dask.array.linalg.svd_compressed**(a, k, n_power_iter=0, seed=None)

Randomly compressed rank-k thin Singular Value Decomposition.

This computes the approximate singular value decomposition of a large array. This algorithm is generally faster than the normal algorithm but does not provide exact results. One can balance between performance and accuracy with input parameters (see below).

Parameters

- **a**: Array Input array
- **k**: int Rank of the desired thin SVD decomposition.
- **n_power_iter**: int Number of power iterations, useful when the singular values decay slowly. Error decreases exponentially as n_power_iter increases. In practice, set n_power_iter <= 4.

Returns

- **u**: Array, unitary / orthogonal
- **s**: Array, singular values in decreasing order (largest first)
- **v**: Array, unitary / orthogonal

References


Examples

```python
>>> u, s, vt = svd_compressed(x, 20)  # doctest: +SKIP
```

**dask.array.linalg.sfqr**(data, name=None)

Direct Short-and-Fat QR

Currently, this is a quick hack for non-tall-and-skinny matrices which are one chunk tall and (unless they are one chunk wide) have chunks that are wider than they are tall
\[ Q [R_1 R_2 \ldots] = [A_1 A_2 \ldots] \]

it computes the factorization \( Q R_1 = A_1 \), then computes the other \( R_k \)'s in parallel.

**Parameters**

- `data`: Array

**See also:**

- `dask.array.linalg.qr`, `dask.array.linalg.tsqr`

`dask.array.linalg.tsqr(data, compute_svd=False, _max_vchunk_size=None)`

Direct Tall-and-Skinny QR algorithm

As presented in:


This algorithm is used to compute both the QR decomposition and the Singular Value Decomposition. It requires that the input array have a single column of blocks, each of which fit in memory.

**Parameters**

- `data`: Array
- `compute_svd`: bool  Whether to compute the SVD rather than the QR decomposition
- `_max_vchunk_size`: Integer  Used internally in recursion to set the maximum row dimension of chunks in subsequent recursive calls.

**See also:**

- `dask.array.linalg.qr`, `dask.array.linalg.svd`, `dask.array.linalg.sfqr`

**Notes**

With \( k \) blocks of size \((m, n)\), this algorithm has memory use that scales as \( k \times m \times n \).

The implementation here is the recursive variant due to the ultimate need for one “single core” QR decomposition. In the non-recursive version of the algorithm, given \( k \) blocks, after \( k \times m \times n \) QR decompositions, there will be a “single core” QR decomposition that will have to work with a \((k \times n, n)\) matrix.

Here, recursion is applied as necessary to ensure that \( k \times n \) is not larger than \( m \) (if \( m / n \geq 2 \)). In particular, this is done to ensure that single core computations do not have to work on blocks larger than \((m, n)\).

Where blocks are irregular, the above logic is applied with the “height” of the “tallest” block used in place of \( m \). Consider use of the `rechunk` method to control this behavior. Blocks that are as tall as possible are recommended.

`dask.array.ma.average(a, axis=None, weights=None, returned=False)`

Return the weighted average of array over the given axis.

**Parameters**

- `a` [array_like]  Data to be averaged. Masked entries are not taken into account in the computation.
- `axis` [int, optional]  Axis along which to average `a`. If `None`, averaging is done over the flattened array.
weights [array_like, optional] The importance that each element has in the computation of the average. The weights array can either be 1-D (in which case its length must be the size of a along the given axis) or of the same shape as a. If weights=None, then all data in a are assumed to have a weight equal to one. If weights is complex, the imaginary parts are ignored.

returned [bool, optional] Flag indicating whether a tuple (result, sum of weights) should be returned as output (True), or just the result (False). Default is False.

Returns

average, [sum_of_weights] [(tuple of) scalar or MaskedArray] The average along the specified axis. When returned is True, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is np.float64 if a is of integer type and floats smaller than float64, or the input data-type, otherwise. If returned, sum_of_weights is always float64.

Examples

```python
>>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
>>> np.ma.average(a, weights=[3, 1, 0, 0])
1.25

>>> x = np.ma.arange(6.).reshape(3, 2)
>>> print(x)
[[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]]
>>> avg, sumweights = np.ma.average(x, axis=0, weights=[1, 2, 3], ... returned=True)
>>> print(avg)
[2.66666666667 3.66666666667]
```

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

```python
>>> 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**

```python
def fill_tangled(n):
    return fill([fill(0) for i in range(n)]
```

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**

4.7. Array 235
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)

masked_array(data =
[[1 --]
[3 4]],

        mask =
[[False True]
[False False]],

    fill_value=999999)

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)

(continues on next page)
mask =
[[False, True],
[False, False]],
fill_value=999999)
>>> ma.getmaskarray(a)
array([[False, True],
       [False, False]])

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]])

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 discontiguous), 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

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
mixed_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

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
mixed_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

```python
>>> 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 <= 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

```python
>>> 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.
```  

```python
>>> 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:

4.7. Array
**masked_where** Mask where a condition is met.

### Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=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 < value)`.

See also:

**masked_where** Mask where a condition is met.

### Examples

```python
>>> 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 <= value)`.

See also:

**masked_where** Mask where a condition is met.

### Examples

```python
>>> 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 != value).

See also:

**masked_where** Mask where a condition is met.

**Examples**

```python
>>> 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)(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**

```python
>>> 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.

```python
>>> 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, determined using **isclose**. The default tolerances for **masked_values** are the same as those for **isclose**.
For integer types, exact equality is used, in the same way as `masked_equal`.

The fill_value is set to `value` and the mask is set to `nomask` if possible.

**Parameters**

- `x` [array_like] Array to mask.
- `value` [float] Masking value.
- `rtol`, `atol` [float, optional] Tolerance parameters passed on to `isclose`
- `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**

```python
>>> 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.

```python
>>> 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`.

```python
>>> 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)

(continues on next page)
When condition or a contain masked values.

```python
>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
```

```python
masked_array(data = [0 1 -- 3],
            mask = [False False True False],
           fill_value=999999)
```

```python
>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)
```

```python
masked_array(data = [-- 1 2 3],
            mask = [ True False False False],
           fill_value=999999)
```

```python
>>> ma.masked_where(a == 3, b)
```

```python
masked_array(data = [-- 1 -- --],
            mask = [ True False True True],
           fill_value=999999)
```

```python
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

```python
>>> 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)
```

Nothing happens if `a` is not a masked array.

```python
>>> 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.overlap.overlap(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. The `{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

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

>>> x = np.arange(64).reshape((8, 8))
>>> d = da.from_array(x, chunks=(4, 4))
```
dask Documentation, Release 1.2.2

>>> d.chunks
((4, 4), (4, 4))

>>> g = da.overlap.overlap(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,  5,  6,  7,  8,  9, 10],
    [16, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26],
    [24, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34],
    [24, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34],
    [40, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50],
    [24, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34],
    [24, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34],
    [40, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50],
    [56, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66],
    [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
    [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]])

MAP OVERLAP

The function to apply to each extended block

The number of elements that each block should share with its neighbors.
If a tuple or dict then this can be different per axis.

How to handle the boundaries. Values include ‘reflect’, ‘periodic’, ‘nearest’, ‘none’,
or any constant value like 0 or np.nan.

Whether or not to trim depth elements from each block after calling the map function.
Set this to False if your mapping function already does this for you.

Other keyword arguments valid in map_blocks.

Examples

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

>>> x = np.array([1, 1, 2, 3, 3, 2, 1, 1])
>>> x = da.from_array(x, chunks=5)
>>> def derivative(x):
...     return x - np.roll(x, 1)
```python
d = x.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]])
```

dask.array.overlap.trim_internal(x, axes, boundary=None)
Trim sides from each block

This couples well with the overlap operation, which may leave excess data on each block

See also:

dask.array.chunk.trim, dask.array.map_blocks

dask.array.overlap.trim_overlap(x, depth, boundary=None)
Trim sides from each block.

This couples well with the map_overlap operation which may leave excess data on each block.

See also:

dask.array.overlap.map_overlap

dask.array.from_array(x, chunks='auto', 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 block-size like 1000. - A blockshape like (1000, 1000). - Explicit sizes of all blocks along all dimensions like
  
  
  ((1000, 1000, 500), (400, 400)).

  - A size in bytes, like “100 MiB” which will choose a uniform block-like shape
  - The word “auto” which acts like the above, but uses a configuration value array.chunk-size for the chunk size

  -1 or None as a blocksize indicate the size of the corresponding dimension.
**name** [str, optional] The key name to use for the array. Defaults to a hash of \( x \). By default, hash uses python’s standard sha1. This behaviour can be changed by installing cityhash, xhash or murmurhash. If installed, a large-factor speedup can be obtained in the tokenisation step. 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 \( x \) doesn’t support fancy indexing (e.g. indexing with lists or arrays) then set to False. Default is True.

### Examples

```python
>>> x = h5py.File('...')['/data/path']  # doctest: +SKIP
>>> a = da.from_array(x, chunks=(1000, 1000))  # doctest: +SKIP

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.

```python
>>> a = da.from_array(x, chunks=(1000, 1000), lock=True)  # doctest: +SKIP

If your underlying datastore has a .chunks attribute (as h5py and zarr datasets do) then a multiple of that chunk shape will be used if you do not provide a chunk shape.

```python
>>> a = da.from_array(x, chunks='auto')  # doctest: +SKIP
>>> a = da.from_array(x, chunks='100 MiB')  # doctest: +SKIP
>>> a = da.from_array(x)  # doctest: +SKIP

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

```python
>>> 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.from_zarr(url, component=None, storage_options=None, chunks=None, name=None, **kwargs)

Load array from the zarr storage format

See https://zarr.readthedocs.io for details about the format.

Parameters

url: Zarr Array or str or MutableMapping Location of the data. A URL can include a protocol specifier like s3:// for remote data. Can also be any MutableMapping instance, which should be serializable if used in multiple processes.

component: str or None If the location is a zarr group rather than an array, this is the subcomponent that should be loaded, something like 'foo/bar'.

storage_options: dict Any additional parameters for the storage backend (ignored for local paths)

chunks: tuple of ints or tuples of ints Passed to da.from_array, allows setting the chunks on initialisation, if the chunking scheme in the on-disc dataset is not optimal for the calculations to follow.

name [str, optional] An optional keyname for the array. Defaults to hashing the input

kwargs: passed to ‘zarr.Array’.

dask.array.from_tiledb(uri, attribute=None, chunks=None, storage_options=None, **kwargs)

Load array from the TileDB storage format

See https://docs.tiledb.io for more information about TileDB.

Parameters

uri: TileDB array or str Location to save the data

attribute: str or None Attribute selection (single-attribute view on multi-attribute array)

Returns

A Dask Array

Examples

```python
>>> # create a tiledb array
>>> import tiledb, numpy as np, tempfile # doctest: +SKIP
>>> uri = tempfile.NamedTemporaryFile().name # doctest: +SKIP
>>> tiledb.from_numpy(uri, np.arange(0,9).reshape(3,3)) # doctest: +SKIP
<tiledb.libtiledb.DenseArray object at 0x...>
>>> # read back the array
>>> import dask.array as da # doctest: +SKIP
>>> tdb_ar = da.from_tiledb(uri) # doctest: +SKIP
>>> tdb_ar.shape # doctest: +SKIP
(3, 3)
>>> tdb_ar.mean().compute() # doctest: +SKIP
4.0
```

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 Delayed or iterable of array-likes and/or Delayeds
- **lock**: boolean or threading.Lock, optional
- **regions**: tuple of slices or list of tuples of slices
- **compute**: boolean, optional
- **return_stored**: boolean, optional

**Examples**

```python
g = dset.data[10:20] = ...  # doctest: +SKIP
```

Alternatively store many arrays at the same time

```python
g = dset.data[10:20] = ...  # doctest: +SKIP
```

**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.

```python
d = da.to_hdf5('myfile.hdf5', '/x', x)  # doctest: +SKIP
```

or

```python
d = da.to_hdf5('myfile.hdf5', '/x': x, '/y': y)  # doctest: +SKIP
```

Optionally provide arguments as though to `h5py.File.create_dataset`
This can also be used as a method on a single Array

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

See also:

dask.store, h5py.File.create_dataset

dask.array.to_zarr(arr, url, component=None, storage_options=None, overwrite=False, compute=True, return_stored=False, **kwargs)

Save array to the zarr storage format

See https://zarr.readthedocs.io for details about the format.

Parameters

- arr: dask.array  Data to store
- url: Zarr Array or str or MutableMapping  Location of the data. A URL can include a protocol specifier like s3:// for remote data. Can also be any MutableMapping instance, which should be serializable if used in multiple processes.
- component: str or None  If the location is a zarr group rather than an array, this is the subcomponent that should be created/over-written.
- storage_options: dict  Any additional parameters for the storage backend (ignored for local paths)
- overwrite: bool  If given array already exists, overwrite=False will cause an error, where overwrite=True will replace the existing data.
- compute, return_stored: see “store()“
- kwargs: passed to the “zarr.create()“ function, e.g., compression options

dask.array.to_npy_stack(dirname, x, axis=0)

Write dask array to a stack of .npy files

This partitions the dask.array along one axis and stores each block along that axis as a single .npy file in the specified directory

See also:

from_npy_stack

Examples

```
>>> x = da.ones((5, 10, 10), chunks=(2, 4, 4))  # doctest: +SKIP
>>> da.to_npy_stack('data/', x, axis=0)  # doctest: +SKIP
```

$ tree data/
data/  
  |  
  |  0.npy  
  |  1.npy  
  |  2.npy  
  |  info

The .npy files store numpy arrays for x[0:2], x[2:4], and x[4:5] respectively, as is specified by the chunk size along the zeroth axis. The info file stores the dtype, chunks, and axis information of the array.

You can load these stacks with the da.from_npy_stack function.

4.7. Array 251
```python
>>> y = da.from_npy_stack('data/')  # doctest: +SKIP
```

dask.array.to_tiledb(darray, uri, compute=True, return_stored=False, storage_options=None, **kwargs)

Save array to the TileDB storage format

Save ‘array’ using the TileDB storage manager, to any TileDB-supported URI, including local disk, S3, or HDFS.

See https://docs.tiledb.io for more information about TileDB.

Parameters

- **darray**: dask.array
  A dask array to write.
- **uri**: Any supported TileDB storage location.
- **storage_options**: dict
  Dict containing any configuration options for the TileDB backend. see https://docs.tiledb.io/en/stable/tutorials/config.html
- **compute**, **return_stored**: see ‘store()’

Returns

- **None** Unless **return_stored** is set to True (False by default)

Notes

TileDB only supports regularly-chunked arrays. TileDB tile extents correspond to form 2 of the dask chunk specification, and the conversion is done automatically for supported arrays.

Examples

```python
>>> import dask.array as da, tempfile  # doctest: +SKIP
>>> uri = tempfile.NamedTemporaryFile().name  # doctest: +SKIP
>>> data = da.random.random(5,5)  # doctest: +SKIP
>>> da.to_tiledb(data, uri)  # doctest: +SKIP
>>> import tiledb  # doctest: +SKIP
>>> tdb_ar = tiledb.open(uri)  # doctest: +SKIP
>>> all(tdb_ar == data)  # doctest: +SKIP
True
```

dask.array.fft.fft_wrap(fft_func, kind=None, dtype=None)

Wrap 1D, 2D, and ND real and 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
- fft2
- fftn
- ifft
- ifft2
- ifftn
Examples

```python
>>> 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.fft

The axis along which the FFT is applied must have a one chunk. To change the array’s chunking use dask.Array.rechunk.

The numpy.fft.fft docstring follows below:

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional $n$-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

Parameters

- **a** [array_like] Input array, can be complex.
- **n** [int, optional] Length of the transformed axis of the output. If $n$ is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If $n$ is not given, the length of the input along the axis specified by `axis` is used.
- **axis** [int, optional] Axis over which to compute the FFT. If not given, the last axis is used.
- **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**

```python
>>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
a = array([-0.00000000e+00 +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])
```

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:

```python
>>> 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)
```

**dask.array.fft.fft2(a, s=None, axes=None)**

Wrapping of `numpy.fft.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.fft2` docstring follows below:

**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.

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 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

```python
>>> 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 ]],
       dtype=complex)
```

(continues on next page)
dask.array.fft.fftn(a, s=None, axes=None)

Wrapping of numpy.fft.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.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 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. 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.
numpy.ifftn The inverse of fftn, the inverse n-dimensional FFT.
numpy.fft The one-dimensional FFT, with definitions and conventions used.
numpy.rfftn The n-dimensional FFT of real input.
numpy.fft2 The two-dimensional FFT.
numpy.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

```python
>>> 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 .``i.fft`` (a, n=None, axis=None)

Wrapping of `numpy.fft.ifft`

The axis along which the FFT is applied must have a one chunk. To change the array’s chunking use `dask.Array.rechunk`.

The `numpy.fft.ifft` docstring follows below:

Compute the one-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the one-dimensional n-point discrete Fourier transform computed by `fft`. In other words, `ifft(fft(a)) == a` to within numerical accuracy. For a general description of the algorithm and definitions, see `numpy.fft`.

The input should be ordered in the same way as it 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 \texttt{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 \texttt{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**

- \texttt{IndexError} If \(axes\) is larger than the last axis of \(a\).

**See also:**

- \texttt{numpy.fft} An introduction, with definitions and general explanations.
- \texttt{fft} The one-dimensional (forward) FFT, of which \texttt{ifft} is the inverse
- \texttt{ifft2} The two-dimensional inverse FFT.
- \texttt{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 \texttt{ifft}.

**Examples**

```python
cimport matplotlib.pyplot as plt
c
t = np.arange(400)
cn = np.zeros((400,), dtype=complex)
cn[40:60] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20,)))
cs = np.fft.ifft(cn)
plt.plot(t, cs.real, 'b-', t, cs.imag, 'r--')
...plt.legend(['real', 'imaginary'])
```

Create and plot a band-limited signal with random phases:
dask.array.fft.ifft2(a, s=\text{None}, \text{axes}=\text{None})

Wrapping of numpy.fft.ifft2

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use \text{dask.Array.rechunk}.

The numpy.fft.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 \text{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

\begin{itemize}
\item \texttt{a} \ [\text{array_like}] Input array, can be complex.
\item \texttt{s} \ [\text{sequence of ints}, \text{optional}] Shape (length of each axis) of the output \texttt{(s[0]} refers to axis 0,
\texttt{s[1]} to axis 1, etc.). This corresponds to \texttt{n} for \texttt{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. \texttt{if} \texttt{s} is not given, the shape of the input along the axes specified by \texttt{axes}
is used. See notes for issue on \texttt{ifft zero padding}.
\item \texttt{axes} \ [\text{sequence of ints}, \text{optional}] Axes over which to compute the FFT. If not given, the last two
axes are used. A repeated index in \texttt{axes} means the transform over that axis is performed
multiple times. A one-element sequence means that a one-dimensional FFT is performed.
\item \texttt{norm} \ [\{\text{None}, “ortho”\}, \text{optional}] New in version 1.10.0.
Normalization mode (see \texttt{numpy.fft}). Default is \texttt{None}.
\end{itemize}

Returns

\begin{itemize}
\item \texttt{out} \ [\text{complex ndarray}] The truncated or zero-padded input, transformed along the axes indicated
by \texttt{axes}, or the last two axes if \texttt{axes} is not given.
\end{itemize}

Raises

\begin{itemize}
\item \texttt{ValueError} \text{ If } \texttt{s} \text{ and } \texttt{axes} \text{ have different length, or } \texttt{axes} \text{ not given and } \text{len}(\texttt{s}) \neq 2.
\item \texttt{IndexError} \text{ If an element of } \texttt{axes} \text{ is larger than than the number of axes of } \texttt{a}.
\end{itemize}

See also:

\begin{itemize}
\item \texttt{numpy.fft} \text{ Overall view of discrete Fourier transforms, with definitions and conventions used.}
\item \texttt{fft2} \text{ The forward 2-dimensional FFT, of which ifft2 is the inverse.}
\item \texttt{ifftn} \text{ The inverse of the } \texttt{n}-\text{dimensional FFT.}
\item \texttt{fft} \text{ The one-dimensional FFT.}
\item \texttt{ifft} \text{ The one-dimensional inverse FFT.}
\end{itemize}
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

```python
>>> 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, 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]])
```

dask.array.fft.ifftn(a, s=None, axes=None)

Wrapping of numpy.fft.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.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(a, 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 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

```python
>>> 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:

```python
>>> 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.rfft**

The axis along which the FFT is applied must have a one chunk. To change the array’s chunking use `dask.Array.rechunk`.

The **numpy.fft.rfft** docstring follows below:

Compute the one-dimensional discrete Fourier Transform for real input.

This function computes the one-dimensional `n`-point discrete Fourier Transform (DFT) of a real-valued array by means of an efficient algorithm called the Fast Fourier Transform (FFT).

**Parameters**

- `a` [array_like] Input array

4.7. 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.


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

```python
>>> np.fft.fft([0, 1, 0, 0])
a = array([ 1.+0.j, 0.-1.j, -1.+0.j, 0.+1.j])
>>> np.fft.rfft([0, 1, 0, 0])
a = 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.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.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.

  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.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.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 rfft(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.

  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

```python
>>> 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],
        [ 0.+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.irfft

The axis along which the FFT is applied must have a one chunk. To change the array's chunking use dask.Array.rechunk.

The numpy.fft.irfft docstring follows below:

Compute the inverse of the n-point DFT for real input.
This function computes the inverse of the one-dimensional \(n\)-point discrete Fourier Transform of real input computed by \(rfft\). In other words, \(\text{irfft}(\text{rfft}(a), \text{len}(a)) == a\) to within numerical accuracy. (See Notes below for why \(\text{len}(a)\) 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 \(\text{irfft}\) 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:

\[a_{\text{resamp}} = \text{irfft} (\text{rfft} (a), m)\]
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.

```python
>>> np.fft.ifft([1, -1j, -1, 1j])
array([ 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j])
```

```python
>>> np.fft.irfft([1, -1j, -1])
array([ 0., 1., 0., 0.])
```

dask.array.fft.irfft2
Wrapping of numpy.fft.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.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
Wrapping of numpy.fft.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.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 `ifft`, 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.
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.


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 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

```python
>>> 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.]],
      [[ 1.,  1.],
       [ 1.,  1.]],
      [[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]],)
```
dask.array.fft.hfft (a, n=None, axis=None)

Wrapping of numpy.fft.hfft

The axis along which the FFT is applied must have a one chunk. To change the array’s chunking use dask.Array.rechunk.

The numpy.fft.hfft docstring follows below:

Compute the FFT of a signal that has Hermitian symmetry, i.e., a real spectrum.

Parameters

a [array_like] The input array.

n [int, optional] Length of the transformed axis of the output. For n output points,
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] Normalization mode (see numpy.fft). Default is None.

New in version 1.10.0.

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 - 2 where m is the length of the transformed axis of the input. To get an odd
number of output points, n must be specified, for instance as 2*m - 1 in the typical case,

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.

• even: ihfft(hfft(a, 2*len(a) - 2) == a, within roundoff error,
• odd: ihfft(hfft(a, 2*len(a) - 1) == a, within roundoff error.

Examples

```python
>>> 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., -1., 0., -4.])
>>> np.fft.hfft(signal, 6) # Input entire signal and truncate
array([ 15., -4., -1., 0., -4.])
```
```python
>>> 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.ihfft

The axis along which the FFT is applied must have a one chunk. To change the array’s chunking use dask.Array.rechunk.

The numpy.fft.ihfft docstring follows below:

Compute the inverse FFT of a signal that has Hermitian symmetry.

**Parameters**

- **a** [array_like] Input array.
- **n** [int, optional] Length of the inverse FFT, the 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] 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. The length of the transformed axis is n//2 + 1.

**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:

- even: ihfft(hfft(a, 2*len(a) - 2) == a, within roundoff error,
- odd: ihfft(hfft(a, 2*len(a) - 1) == a, within roundoff error.

**Examples**
```python
>>> 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**

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 \( d \):

\[
f = \begin{cases} 
[0, 1, \ldots, \frac{n}{2}-1, -\frac{n}{2}, \ldots, -1] / (d \times n) & \text{if } n \text { is even} \\
[0, 1, \ldots, \frac{(n-1)}{2}, -\frac{(n-1)}{2}, \ldots, -1] / (d \times n) & \text{if } n \text { is odd}
\end{cases}
\]

**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**

```python
>>> 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**

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 \( d \):

\[
f = \begin{cases} 
[0, 1, \ldots, \frac{n}{2}-1, \frac{n}{2}] / (d \times n) & \text{if } n \text { is even} \\
[0, 1, \ldots, \frac{(n-1)}{2}, \frac{(n-1)}{2}] / (d \times n) & \text{if } n \text { is odd}
\end{cases}
\]

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

```python
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

```python
>>> 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:

```python
>>> 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

```python
>>> 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 or array_like of floats] Alpha, positive (>0).
- b [float or array_like of floats] Beta, positive (>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. If size is None (default), a single value is returned if a and b are both scalars. Otherwise, np.broadcast(a, b).size samples are drawn.

Returns

- out [ndarray or scalar] Drawn samples from the parameterized beta distribution.

dask.array.random.binomial(n, p, size=None)

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)
Parameters

- **n** [int or array_like of ints] Parameter of the distribution, \( \geq 0 \). Floats are also accepted, but they will be truncated to integers.
- **p** [float or array_like of floats] Parameter of the distribution, \( \geq 0 \) and \( \leq 1 \).
- **size** [int or tuple of ints, optional] Output shape. If the given shape is, e.g., \((m, n, k)\), then \( m \times n \times k \) samples are drawn. If size is \texttt{None} (default), a single value is returned if \( n \) and \( p \) are both scalars. Otherwise, \texttt{np.broadcast(n, p).size} samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the \( n \) trials.

See also:

- `scipy.stats.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

[1], [2], [3], [4], [5]

Examples

Draw samples from the distribution:

```python
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000) # doctest: +SKIP
# 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.

```python
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000. # doctest: +SKIP
# 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 [float or array_like of floats] Number of 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. If size is None (default), a single value is returned if df
  is a scalar. Otherwise, np.array(df).size samples are drawn.

Returns

- out [ndarray or scalar] Drawn samples from the parameterized chi-square distribution.

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 vari-
ables:

\[ Q = \sum_{i=0}^{df} X_i^2 \]

is chi-square distributed, denoted

\[ Q \sim \chi^2_k. \]

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

[1]

Examples

```python
>>> np.random.chisquare(2, 4)  # doctest: +SKIP
array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```
New in version 1.7.0.

Parameters

- **a** [1-D array-like or int]
  If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a were np.arange(a)

- **size** [int or tuple of ints, optional] Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn. Default is None, in which case a single value is returned.

- **replace** [boolean, optional] Whether the sample is with or without replacement

- **p** [1-D array-like, optional] The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

Returns

- **samples** [single item or ndarray] The generated random samples

Raises

**ValueError** If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

Generate a non-uniform random sample from np.arange(5) of size 3:

```python
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])  # doctest: +SKIP
array([3, 3, 0])
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```python
>>> np.random.choice(5, 3, replace=False)  # doctest: +SKIP
array([3, 1, 0])
```

```python
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```python
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])  # doctest: +SKIP
array([2, 3, 0])
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```python
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']  # doctest: +SKIP
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])  # doctest: +SKIP
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'],
      dtype='|S11')
```
dask.array.random.exponential(scale=1.0, size=None)

Draw samples from an exponential distribution.

Its probability density function is

\[ f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right), \]

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 [3].

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 [1], or the time between page requests to Wikipedia [2].

Parameters

- **scale** [float or array_like of floats] 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 \(m \times n \times k\) samples are drawn. If size is None (default), a single value is returned if scale is a scalar. Otherwise, np.array(scale).size samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized exponential distribution.

References

[1], [2], [3]

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 or array_like of floats] Degrees of freedom in numerator, should be > 0.
- **dfden** [float or array_like of float] Degrees of freedom in denominator, should be > 0.
- **size** [int or tuple of ints, optional] Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn. If size is None (default), a single value is returned if dfnum and dfden are both scalars. Otherwise, np.broadcast(dfnum, dfden).size samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized Fisher distribution.

See also:

- scipy.stats.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

[1], [2]

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  # doctest: +SKIP
>>> dfden = 48. # within groups degrees of freedom  # doctest: +SKIP
>>> s = np.random.f(dfnum, dfden, 1000)  # doctest: +SKIP
```

The lower bound for the top 1% of the samples is:

```
>>> sort(s)[-10]  # doctest: +SKIP
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, \( \text{shape} \) (sometimes designated “\( k \)”) and \( \text{scale} \) (sometimes designated “\( \theta \)”), where both parameters are > 0.

Parameters

- **shape** [float or array_like of floats] The shape of the gamma distribution. Should be greater than zero.

- **scale** [float or array_like of floats, optional] The scale of the gamma distribution. Should be greater than zero. 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 \(m \times n \times k\) samples are drawn. If size is None (default), a single value is returned if shape and scale are both scalars. Otherwise, np.broadcast(shape, scale). size samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized gamma distribution.

See also:
scipy.stats.gamma  probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

\[ p(x) = x^{k-1} e^{-x/\theta} \frac{1}{\theta^k \Gamma(k)}, \]

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

[1], [2]

Examples

Draw samples from the distribution:

```python
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)  # doctest: +SKIP
>>> s = np.random.gamma(shape, scale, 1000)  # doctest: +SKIP
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> import scipy.special as sps  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, 50, density=True)  # doctest: +SKIP
>>> y = bins**(shape-1)*(np.exp(-bins/scale) / (sps.gamma(shape)*scale**shape))  # doctest: +SKIP
>>> plt.plot(bins, y, linewidth=2, color='r')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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, \( k = 1, 2, \ldots \).

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 or array_like of floats] 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 \( m \times n \times k \) samples are drawn. If size is None (default), a single value is returned if p is a scalar. Otherwise, np.ndarray(p).size samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized geometric distribution.
Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```python
>>> z = np.random.geometric(p=0.35, size=10000)  # doctest: +SKIP
```

How many trials succeeded after a single run?

```python
>>> (z == 1).sum() / 10000.  # doctest: +SKIP
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 or array_like of floats, optional] The location of the mode of the distribution. Default is 0.
- `scale` [float or array_like of floats, optional] The scale parameter of the distribution. Default is 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. If `size` is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns

- `out` [ndarray or scalar] Drawn samples from the parameterized Gumbel distribution.

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

$$ p(x) = \frac{e^{-\frac{x-\mu}{\beta}}}{\beta} e^{-e^{-\frac{x-\mu}{\beta}}} , $$

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 Frechet.
The function has a mean of $\mu + 0.57721 \beta$ and a variance of $\frac{\pi^2}{6} \beta^2$.

References

[1], [2]

Examples

Draw samples from the distribution:

```python
>>> mu, beta = 0, 0.1 # location and scale # doctest: +SKIP
>>> s = np.random.gumbel(mu, beta, 1000) # doctest: +SKIP

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, 30, density=True) # doctest: +SKIP
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta) * np.exp(-np.exp(-(bins - mu)/beta)), linewidth=2, color='r')
>>> plt.show() # doctest: +SKIP
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```python
>>> means = [] # doctest: +SKIP
>>> maxima = [] # doctest: +SKIP
>>> for i in range(0,1000): # doctest: +SKIP
...     a = np.random.normal(mu, beta, 1000)
...     means.append(a.mean())
...     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, density=True) # doctest: +SKIP
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi # doctest: +SKIP
>>> mu = np.mean(maxima) - 0.57721*beta # doctest: +SKIP
>>> 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() # doctest: +SKIP
```

```python
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 of ints] Number of ways to make a good selection. Must be nonnegative.
- **nbad** [int or array_like of ints] Number of ways to make a bad selection. Must be nonnegative.
- **nsample** [int or array_like of ints] 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. If size is None (default), a single value is returned if ngood, nbad, and nsample are all scalars. Otherwise, np.broadcast(ngood, nbad, nsample).size samples are drawn.

Returns
out [ndarray or scalar] Drawn samples from the parameterized hypergeometric distribution.

See also:
scipy.stats.hypergeom probability density function, distribution or cumulative density function, etc.

Notes
The probability density for the Hypergeometric distribution is
\[ P(x) = \frac{\binom{g}{x} \binom{b}{n-x}}{\binom{g+b}{n}}, \]
where 0 ≤ x ≤ n and n − b ≤ x ≤ g
for P(x) the probability of x successes, g = ngood, b = nbad, and n = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

References
[1], [2], [3]

Examples
Draw samples from the distribution:

```python
>>> ngood, nbad, nsamp = 100, 2, 10  # doctest: +SKIP
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)  # doctest: +SKIP
>>> from matplotlib.pyplot import hist  # doctest: +SKIP
>>> hist(s)  # doctest: +SKIP
# 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?

```python
>>> s = np.random.hypergeometric(15, 15, 15, 100000)  # doctest: +SKIP
>>> sum(s>=12)/100000. + sum(s<=3)/100000.  # doctest: +SKIP
# 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 or array_like of floats, optional] The position, \( \mu \), of the distribution peak. Default is 0.
- `scale` [float or array_like of floats, optional] \( \lambda \), the exponential decay. Default is 1.
- `size` [int or tuple of ints, optional] Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn. If `size` is None (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, np.broadcast(loc, scale).size samples are drawn.

**Returns**

- `out` [ndarray or scalar] Drawn samples from the parameterized Laplace distribution.

**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**

[1], [2], [3], [4]

**Examples**

Draw samples from the distribution

```python
>>> loc, scale = 0., 1.  # doctest: +SKIP
>>> s = np.random.laplace(loc, scale, 1000)  # doctest: +SKIP
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, 30, density=True)  # doctest: +SKIP
>>> x = np.arange(-8., 8., .01)  # doctest: +SKIP
>>> pdf = np.exp(-np.abs(x-loc)/scale)/(2.*scale)  # doctest: +SKIP
>>> plt.plot(x, pdf)  # doctest: +SKIP
```

Plot Gaussian for comparison:

```python
>>> g = (1/(scale * np.sqrt(2 * np.pi))) * np.exp(-((x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x, g)  # doctest: +SKIP
```
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 or array_like of floats, optional] Parameter of the distribution. Default is 0.

scale [float or array_like of floats, optional] Parameter of the distribution. Should be greater than zero. Default is 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. If size is None (default), a single value is returned if loc and scale are both scalars. Otherwise, np.broadcast(loc, scale).size samples are drawn.

Returns

out [ndarray or scalar] Drawn samples from the parameterized logistic distribution.

See also:

scipy.stats.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

[1], [2], [3]

Examples

Draw samples from the distribution:

```python
>>> loc, scale = 10, 1  # doctest: +SKIP
>>> s = np.random.logistic(loc, scale, 10000)  # doctest: +SKIP
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, bins=50)  # doctest: +SKIP
>>> # plot against distribution
```
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 or array_like of floats, optional]

Mean value of the underlying normal distribution. Default is 0.

sigma [float or array_like of floats, optional] Standard deviation of the underlying normal distribution. Should be greater than zero. Default is 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. If size is None (default), a single value is returned if mean and sigma are both scalars. Otherwise, np.broadcast(mean, sigma). size samples are drawn.

Returns

out [ndarray or scalar] Drawn samples from the parameterized log-normal distribution.

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:

\[
p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{\left(\frac{-(\log(x) - \mu)^2}{2\sigma^2}\right)}
\]

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.

>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = [] # doctest: +SKIP
>>> for i in range(1000): # doctest: +SKIP
...    a = 10. + np.random.random(100)
...    b.append(np.product(a))
```python
d = np.array(b) / np.min(b)  # scale values to be positive  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(b, 100, density=True, align='mid')  # doctest: +SKIP
...  # doctest: +SKIP
>>> sigma = np.std(np.log(b))  # doctest: +SKIP
...  # doctest: +SKIP
>>> mu = np.mean(np.log(b))  # doctest: +SKIP
...  # doctest: +SKIP
>>> x = np.linspace(min(bins), max(bins), 10000)  # doctest: +SKIP
...  # doctest: +SKIP
...  # doctest: +SKIP
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))  # doctest: +SKIP
...  # doctest: +SKIP
...  # doctest: +SKIP
>>> plt.plot(x, pdf, color='r', linewidth=2)  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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

- **p** [float or array_like of floats] Shape parameter for the distribution. Must be in the range (0, 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. If size is None (default), a single value is returned if p is
  a scalar. Otherwise, np.array(p).size samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized logarithmic series distribution.

See also:

- scipy.stats.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

[1], [2], [3], [4]

Examples

Draw samples from the distribution:
```python
>>> a = .6  # doctest: +SKIP
>>> s = np.random.logseries(a, 10000)  # doctest: +SKIP
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s)  # doctest: +SKIP

# plot against distribution

>>> def logseries(k, p):  # doctest: +SKIP
...    return -p**k/(k*log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/(logseries(bins, a).max()), 'r')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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 successes and p probability of success where n is an integer > 0 and p is in the interval [0, 1].

Parameters

- **n** [int or array_like of ints] Parameter of the distribution, > 0. Floats are also accepted, but they will be truncated to integers.
- **p** [float or array_like of floats] Parameter of the distribution, >= 0 and <=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. If size is None (default), a single value is returned if n and p are both scalars. Otherwise, np.broadcast(n, p).size samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of failures that occurred before a total of n successes was reached.

Notes

The probability density for the negative binomial distribution is

\[
P(N; n, p) = \binom{N + n - 1}{N} p^n (1-p)^N,
\]

where \( n \) is the number of successes, \( p \) is the probability of success, and \( N + n \) is the number of trials. The negative binomial distribution gives the probability of \( N \) failures given \( n \) successes, with a success on the last 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

[1], [2]
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.?

```python
>>> s = np.random.negative_binomial(1, 0.1, 100000)  # doctest: +SKIP
>>> for i in range(1, 11):  # doctest: +SKIP
...    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` [float or array_like of floats] Degrees of freedom, should be > 0.
  
  Changed in version 1.10.0: Earlier NumPy versions required dfnum > 1.
  
- `nonc` [float or array_like of floats] 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 \(m \times n \times k\) samples are drawn. If size is None (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, np.broadcast(`df`, `nonc`).size samples are drawn.

**Returns**

- `out` [ndarray or scalar] Drawn samples from the parameterized noncentral chi-square distribution.

**Notes**

The probability density function for the noncentral Chi-square distribution is

\[
P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} Y_{df+2i}(x),
\]

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**

[1], [2]

**Examples**

Draw values from the distribution and plot the histogram
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),  # doctest:
...                    bins=200, density=True)
>>> plt.show()  # doctest: +SKIP

Draw values from a noncentral chi-square with very small noncentrality, and compare to a chisquare.

>>> plt.figure()  # doctest: +SKIP
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),  # doctest:
...                    bins=np.arange(0., 25, .1), density=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),  # doctest: +SKIP
...                    bins=np.arange(0., 25, .1), density=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```python
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),  # doctest:
...                    bins=200, density=True)
>>> plt.show()  # doctest: +SKIP
```

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 [float or array_like of floats] Numerator degrees of freedom, should be > 0.
  - Changed in version 1.14.0: Earlier NumPy versions required dfnum > 1.
- dfden [float or array_like of floats] Denominator degrees of freedom, should be > 0.
- nonc [float or array_like of floats] Non-centrality parameter, the sum of the squares of the numerator means, 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. If size is None (default), a single value is returned if dfnum, dfden, and nonc are all scalars. Otherwise, np.broadcast(dfnum, dfden, nonc).size samples are drawn.

Returns

- out [ndarray or scalar] Drawn samples from the parameterized noncentral Fisher distribution.

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
[1], [2]

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.

```python
>>> dfnum = 3  # between group deg of freedom  # doctest: +SKIP
>>> dfden = 20  # within groups degrees of freedom  # doctest: +SKIP
>>> nonc = 3.0  # doctest: +SKIP
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)  # doctest: +SKIP
>>> NF = np.histogram(nc_vals, bins=50, density=True)  # doctest: +SKIP
>>> c_vals = np.random.f(dfnum, dfden, 1000000)  # doctest: +SKIP
>>> F = np.histogram(c_vals, bins=50, density=True)  # doctest: +SKIP
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> plt.plot(F[1][1:], F[0])  # doctest: +SKIP
>>> plt.plot(NF[1][1:], NF[0])  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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 [2], 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 [2].

Parameters
- **loc** [float or array_like of floats] Mean (“centre”) of the distribution.
- **scale** [float or array_like of floats] Standard deviation (spread or “width”) of the distribution.
- **size** [int or tuple of ints, optional] Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn. If size is None (default), a single value is returned if loc and scale are both scalars. Otherwise, np.broadcast(loc, scale).size samples are drawn.

Returns
- **out** [ndarray or scalar] Drawn samples from the parameterized normal distribution.

See also:
- scipy.stats.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 $x - \sigma$ [2]). This implies that numpy.random.normal is more likely to return samples lying close to the mean, rather than those far away.

References

[1], [2]

Examples

Draw samples from the distribution:

```python
>>> mu, sigma = 0, 0.1 # mean and standard deviation # doctest: +SKIP
>>> s = np.random.normal(mu, sigma, 1000) # doctest: +SKIP
```

Verify the mean and the variance:

```python
>>> abs(mu - np.mean(s)) < 0.01 # doctest: +SKIP
True
```

```python
>>> abs(sigma - np.std(s, ddof=1)) < 0.01 # doctest: +SKIP
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, 30, density=True) # doctest: +SKIP
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
... np.exp(- (bins - mu)**2 / (2 * sigma**2)),
... linewidth=2, color='r')
```

```python
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

- **a** [float or array_like of floats] Shape of the distribution. 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. If size is None (default), a single value is returned if a is a scalar. Otherwise, np.array(a).size samples are drawn.

Returns
**`dask.array.random.pareto()`**

Draw samples from the parameterized Pareto distribution.

**Parameters**

- `lam` [float or array_like of floats] Expectation of interval, should be >= 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. If `size` is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

**Returns**

- `out` [ndarray or scalar] Drawn samples from the parameterized Pareto distribution.

**See also:**

- `scipy.stats.lomax` probability density function, distribution or cumulative density function, etc.
- `scipy.stats.genpareto` 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 [1]. It is one of the so-called “fat-tailed” distributions.

**References**

[1], [2], [3], [4]

**Examples**

Draw samples from the distribution:

```python
>>> a, m = 3., 2.  # shape and mode  # doctest: +SKIP
>>> s = (np.random.pareto(a, 1000) + 1) * m  # doctest: +SKIP
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> count, bins, _ = plt.hist(s, 100, density=True)  # doctest: +SKIP
>>> fit = a*m**a / bins**(a+1)  # doctest: +SKIP
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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 array_like of floats] Expectation of interval, should be >= 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. If `size` is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

**Returns**
out [ndarray or scalar] Drawn samples from the parameterized Poisson distribution.

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 \( \text{lam} \) is within 10 sigma of the maximum representable value.

References

[1], [2]

Examples

Draw samples from the distribution:

```python
>>> import numpy as np  # doctest: +SKIP
>>> s = np.random.poisson(5, 10000)  # doctest: +SKIP
```

Display histogram of the sample:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, 14, density=True)  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

Draw each 100 values for lambda 100 and 500:

```python
>>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))  # doctest: +SKIP
```

dask.array.random.power \( a, \text{size=None} \)

Draws samples in \([0, 1]\) from a power distribution with positive exponent \( a - 1 \).

Also known as the power function distribution.

Parameters

- \( a \) [float or array_like of floats] Parameter of the distribution. Should be greater than zero.
- \( \text{size} \) [int or tuple of ints, optional] Output shape. If the given shape is, e.g., \((m, n, k)\), then \( m \times n \times k \) samples are drawn. If \( \text{size} \) is \( \text{None} \) (default), a single value is returned if \( a \) is a scalar. Otherwise, \( \text{np.array}(a).\text{size} \) samples are drawn.

Returns

- \( \text{out} \) [ndarray or scalar] Drawn samples from the parameterized power distribution.

Raises

- \( \text{ValueError} \) If \( a < 1 \).
Notes

The probability density function is

\[ P(x; a) = ax^{a-1}, \quad 0 \leq x \leq 1, \ a > 0. \]

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

[1], [2]

Examples

Draw samples from the distribution:

```python
>>> a = 5.  # shape  # doctest: +SKIP
>>> samples = 1000  # doctest: +SKIP
>>> s = np.random.power(a, samples)  # doctest: +SKIP

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, bins=30)  # doctest: +SKIP
>>> x = np.linspace(0, 1, 100)  # doctest: +SKIP
>>> y = a*x**(a-1.)  # doctest: +SKIP
>>> normed_y = samples*np.diff(bins)[0]*y  # doctest: +SKIP
>>> plt.plot(x, normed_y)  # doctest: +SKIP

Compare the power function distribution to the inverse of the Pareto.

```python
>>> from scipy import stats  # doctest: +SKIP
>>> rvs = np.random.power(5, 1000000)  # doctest: +SKIP
>>> rvsp = np.random.pareto(5, 1000000)  # doctest: +SKIP
>>> xx = np.linspace(0,1,100)  # doctest: +SKIP
>>> powpdf = stats.powerlaw.pdf(xx,5)  # doctest: +SKIP

```python
>>> plt.figure()  # doctest: +SKIP
>>> plt.hist(rvs, bins=50, density=True)  # doctest: +SKIP
>>> plt.plot(xx,powpdf,'r-')  # doctest: +SKIP
>>> plt.title('np.random.power(5)')  # doctest: +SKIP

```python
>>> plt.figure()  # doctest: +SKIP
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)  # doctest: +SKIP
>>> plt.plot(xx,powpdf,'r-')  # doctest: +SKIP
>>> plt.title('inverse of 1 + np.random.pareto(5)')  # doctest: +SKIP

```python
>>> plt.figure()  # doctest: +SKIP
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)  # doctest: +SKIP
>>> plt.plot(xx,powpdf,'r-')  # doctest: +SKIP
>>> plt.title('inverse of stats.pareto(5)')  # doctest: +SKIP
```
dask.array.random.randint (low, high=None, size=None, dtype='l')

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the “discrete uniform” distribution of the specified dtype in the “half-open” interval [low, high). If high is None (the default), then results are from [0, low).

Parameters

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is one above the highest such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

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.

dtype [dtype, optional] Desired dtype of the result. All dtypes are determined by their name, i.e., ‘int64’, ‘int’, etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is ‘np.int’.

New in version 1.11.0.

Returns

out [int or ndarray of ints] size-shaped array of random integers from the appropriate distribution, or a single such random int if size not provided.

See also:

random.random_integers similar to randint, only for the closed interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

Examples

```python
>>> np.random.randint(2, size=10)  # doctest: +SKIP
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```python
>>> np.random.randint(5, size=(2, 4))  # doctest: +SKIP
array([[4, 0, 2, 1], [3, 2, 2, 0]])
```

dask.array.random.random (size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the “continuous uniform” distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of random_sample by (b-a) and add $a$:

$$(b - a) \times \text{random\_sample()} + a$$

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

.. code-block:: python

    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

Three-by-two array of random numbers from [-5, 0):

.. code-block:: python

    5 * np.random.random_sample((3, 2)) - 5

Results are from the “continuous uniform” distribution over the stated interval. To sample $U[a,b), b > a$ multiply the output of random_sample by $(b-a)$ and add $a$:

.. code-block:: python

    (b - a) * random_sample() + a

Parameters

.. code-block:: python

    size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., (m, n, k), then $m \times n \times k$ samples are drawn. Default is None, in which case a single value is returned.

Returns

.. code-block:: python

    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

Three-by-two array of random numbers from [-5, 0):

.. code-block:: python

    5 * np.random.random_sample((3, 2)) - 5
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 [float or array_like of floats, optional] Scale, also equals the mode. Should be $\geq 0$.
        Default is 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. If size is None (default), a single value is returned if scale
        is a scalar. Otherwise, np.array(scale).size samples are drawn.

Returns

out [ndarray or scalar] Drawn samples from the parameterized Rayleigh distribution.

Notes

The probability density function for the Rayleigh distribution is

$$ P(x; \text{scale}) = \frac{x}{\text{scale}^2} e^{-(x/\text{scale})^2} $$

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

[1], [2]

Examples

Draw values from the distribution and plot the histogram

```python
>>> from matplotlib.pyplot import hist
# doctest: +SKIP
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, density=True)  # doctest: +SKIP
```

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?

```python
>>> meanvalue = 1  # doctest: +SKIP
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue  # doctest: +SKIP
>>> s = np.random.rayleigh(modevalue, 1000000)  # doctest: +SKIP
```

The percentage of waves larger than 3 meters is:

```python
>>> 100.*sum(s>3)/1000000.  # doctest: +SKIP
0.08730000000000003
```

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

[1], [2], [3]

Examples

Draw samples and plot the distribution:

```python
>>> import matplotlib.pyplot as plt
>>> s = np.random.standard_cauchy(1000000) # doctest: +SKIP
>>> s = s[(s>-25) & (s<25)] # truncate distribution so it plots well # doctest: +SKIP
>>> plt.hist(s, bins=100) # doctest: +SKIP
>>> plt.show() # doctest: +SKIP
```

`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:

```python
g = np.random.standard_exponential((3, 8000))  # doctest: +SKIP
```

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 or array_like of floats] 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. If size is None (default), a single value is returned if shape is a scalar. Otherwise, np.array(shape).size samples are drawn.

Returns

- **out** [ndarray or scalar] Drawn samples from the parameterized standard gamma distribution.

See also:

- scipy.stats.gamma probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

\[ p(x) = x^{k-1} e^{-x/\theta} / \theta^k \Gamma(k), \]

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

[1], [2]

Examples

Draw samples from the distribution:

```python
>>> shape, scale = 2., 1.  # mean and width  # doctest: +SKIP
>>> s = np.random.standard_gamma(shape, 1000000)  # doctest: +SKIP
```

Display the histogram of the samples, along with the probability density function:
```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> import scipy.special as sps  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, 50, density=True)  # doctest: +SKIP
>>> y = bins**(shape-1) * ((np.exp(-bins/scale)) / 
... (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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**

```python
>>> s = np.random.standard_normal(8000)  # doctest: +SKIP
>>> s  # doctest: +SKIP
array([-0.6888893 , 0.78096262, -0.89086505, ..., 0.49876311, #random
    -0.38672696, -0.4685006 ])
>>> s.shape  # doctest: +SKIP
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))  # doctest: +SKIP
>>> s.shape  # doctest: +SKIP
(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** [float or array_like of floats] 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. If size is None (default), a single value is returned if df is a scalar. Otherwise, np.array(df).size samples are drawn.

**Returns**

- **out** [ndarray or scalar] Drawn samples from the parameterized standard Student’s t distribution.

**Notes**

The probability density function for the t distribution is

\[ P(x, df) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi df} \Gamma\left(\frac{df}{2}\right)} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2} \]

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 Gosset 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
[1], [2]

Examples
From Dalgaard page 83 [1], suppose the daily energy intake for 11 women in kilojoules (kJ) is:

```python
>>> 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?

```python
>>> s = np.random.standard_t(10, size=100000) # doctest: +SKIP
>>> np.mean(intake) # doctest: +SKIP
6753.636363636364
>>> intake.std(ddof=1) # doctest: +SKIP
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.

```python
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake))) # doctest: +SKIP
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```python
>>> np.sum(s<t) / float(len(s)) # doctest: +SKIP
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

`dask.array.random.triangular(left, mode, right, size=None)`

Draw samples from the triangular distribution over the interval [left, right]. 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 [float or array_like of floats] Lower limit.

mode [float or array_like of floats] The value where the peak of the distribution occurs. The value should fulfill the condition left <= mode <= right.

right [float or array_like of floats] 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. If size is None (default), a single value is returned if
left, mode, and right are all scalars. Otherwise, `np.broadcast(left, mode, right).size` samples are drawn.

Returns

out [ndarray or scalar] Drawn samples from the parameterized triangular distribution.

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

[1]

Examples

Draw values from the distribution and plot the histogram:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
... h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200, density=True)
... plt.show()  # doctest: +SKIP
```

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 or array_like of floats, 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 or array_like of floats] 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. If size is None (default), a single value is returned if low and high are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns

out [ndarray or scalar] Drawn samples from the parameterized uniform distribution.

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., \(\text{rand}(2, 2)\) 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.

When \(high == low\), values of \(low\) will be returned. If \(high < low\), the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

**Examples**

Draw samples from the distribution:

```python
>>> s = np.random.uniform(-1,0,1000)  # doctest: +SKIP
```

All values are within the given interval:

```python
>>> np.all(s >= -1)  # doctest: +SKIP
True
>>> np.all(s < 0)    # doctest: +SKIP
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> count, bins, ignored = plt.hist(s, 15, density=True)  # doctest: +SKIP
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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 or array_like of floats] Mode (“center”) of the distribution.
- **kappa** [float or array_like of floats] Dispersion of the distribution, has to 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. If size is None (default), a single value is returned if mu and kappa are both scalars. Otherwise, np.broadcast(mu, kappa).size samples are drawn.

Returns

out [ndarray or scalar] Drawn samples from the parameterized von Mises distribution.

See also:

scipy.stats.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

[1], [2]

Examples

Draw samples from the distribution:

```python
>>> mu, kappa = 0.0, 4.0  # mean and dispersion  # doctest: +SKIP
>>> s = np.random.vonmises(mu, kappa, 1000)  # doctest: +SKIP
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> from scipy.special import i0  # doctest: +SKIP
>>> x = np.linspace(-np.pi, np.pi, num=51)  # doctest: +SKIP
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))  # doctest: +SKIP
>>> plt.plot(x, y, linewidth=2, color='r')  # doctest: +SKIP
>>> plt.hist(s, 50, density=True)  # doctest: +SKIP
>>> plt.show()  # doctest: +SKIP
```

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** [float or array_like of floats] Distribution mean, should be > 0.

**scale** [float or array_like of floats] 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. If size is None (default), a single value is returned if mean and scale are both scalars. Otherwise, np.broadcast(mean, scale). size samples are drawn.

Returns

**out** [ndarray or scalar] Drawn samples from the parameterized Wald distribution.

Notes

The probability density function for the Wald distribution is

\[ P(x; \text{mean}, \text{scale}) = \sqrt{\frac{\text{scale}}{2\pi x^3}} e^{-\text{scale}(x - \text{mean})^2/2} \cdot \frac{\text{mean}}{x} \]

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

[1], [2], [3]

Examples

Draw values from the distribution and plot the histogram:

```python
>>> import matplotlib.pyplot as plt
# doctest: +SKIP
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, density=True) # doctest: +SKIP
>>> plt.show() # doctest: +SKIP
```

dask.array.random.wald(a, size=None)

Draw samples from a Wald distribution.

Draw samples from a 1-parameter Wald 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 Wald, including a scale parameter \(\lambda\) is just \(X = \lambda(-\ln(U))^{1/a}\).

Parameters

**a** [float or array_like of floats] Shape parameter of the distribution. Must be nonnegative.

**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. If size is None (default), a single value is returned if a is a scalar. Otherwise, np.array(a).size samples are drawn.

Returns

304 Chapter 4. Index
out [ndarray or scalar] Drawn samples from the parameterized Weibull distribution.

See also:
scipy.stats.weibull_max, scipy.stats.weibull_min, scipy.stats.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

\[ p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-\left(\frac{x}{\lambda}\right)^a}, \]

where \( a \) is the shape and \( \lambda \) the scale.

The function has its peak (the mode) at \( \lambda^{(a-1)/a} \).

When \( a = 1 \), the Weibull distribution reduces to the exponential distribution.

References

[1], [2], [3]

Examples

Draw samples from the distribution:

```python
>>> a = 5.  # shape  # doctest: +SKIP
>>> s = np.random.weibull(a, 1000)  # doctest: +SKIP
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt  # doctest: +SKIP
>>> x = np.arange(1,100.)/50.  # doctest: +SKIP
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(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))  # doctest: +SKIP
>>> x = np.arange(1,100.)/50.  # doctest: +SKIP
```

```python
dask.array.random.zipf(a, size=None)  # doctest: +SKIP
```

```
```
The arrays must have the same shape, except in the dimension corresponding to \textit{axis} (the first, by default).

\texttt{axis \ [int or None, optional]} Axis along which to compute test. If None, compute over the whole arrays, \texttt{a}, and \texttt{b}.

\texttt{equal\_var \ [bool, optional]} If True (default), perform a standard independent 2 sample test that assumes equal population variances [1]. If False, perform Welch’s t-test, which does not assume equal population variance [2].

New in version 0.11.0.

\texttt{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’.

\textbf{Returns}

\texttt{statistic \ [float or array]} The calculated t-statistic.

\texttt{pvalue \ [float or array]} The two-tailed p-value.

\textbf{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.

\textbf{Examples}

```python
>>> from scipy import stats  # doctest: +SKIP
>>> np.random.seed(12345678)  # doctest: +SKIP

Test with sample with identical means:

```python
>>> rvs1 = stats.norm.rvs(loc=5, scale=10, size=500)  # doctest: +SKIP
>>> rvs2 = stats.norm.rvs(loc=5, scale=10, size=500)  # doctest: +SKIP
>>> stats.ttest_ind(rvs1, rvs2)  # doctest: +SKIP
(0.26833823296239279, 0.78849443369564776)
>>> stats.ttest_ind(rvs1, rvs2, equal_var = False)  # doctest: +SKIP
(0.26833823296239279, 0.78849452749500748)
```

\texttt{ttest\_ind} underestimates \textit{p} for unequal variances:

```python
>>> rvs3 = stats.norm.rvs(loc=5, scale=20, size=500)  # doctest: +SKIP
>>> stats.ttest_ind(rvs1, rvs3)  # doctest: +SKIP
(-0.46580283296239279, 0.64145827413436174)
>>> stats.ttest_ind(rvs1, rvs3, equal_var = False)  # doctest: +SKIP
(-0.46580283296239279, 0.64149646246569292)
```

When \texttt{n1} \neq \texttt{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)  # doctest: +SKIP
>>> stats.ttest_ind(rvs1, rvs4)  # doctest: +SKIP
(-0.99882539442782481, 0.3182832709103896)
>>> stats.ttest_ind(rvs1, rvs4, equal_var = False)  # doctest: +SKIP
(-0.69712570584654099, 0.48716927725402048)

T-test with different means, variance, and n:

>>> rvs5 = stats.norm.rvs(loc=8, scale=20, size=100)  # doctest: +SKIP
>>> stats.ttest_ind(rvs1, rvs5)  # doctest: +SKIP
(-1.4679669854490653, 0.14263895620529152)
>>> stats.ttest_ind(rvs1, rvs5, equal_var = False)  # doctest: +SKIP
(-0.94365973617132992, 0.34744170334794122)

dask.array.stats.ttest_1samp(a, popmean, axis=0, nan_policy='propagate')
Calculate 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, then 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  # doctest: +SKIP

>>> np.random.seed(7654567)  # fix seed to get the same result  # doctest: +SKIP
>>> rvs = stats.norm.rvs(loc=5, scale=10, size=(50,2))  # doctest: +SKIP

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)  # doctest: +SKIP
(array([-0.68014479, -0.04323899]), array([ 0.49961383, 0.96568674]))
>>> stats.ttest_1samp(rvs,0.0)  # doctest: +SKIP
(array([ 2.77025808, 4.11038784]), array([ 0.00789095, 0.00014999]))

Examples using axis and non-scalar dimension for population mean.
dask.array.stats.ttest_rel(a, b, axis=0, nan_policy='propagate')

Calculate 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**

https://en.wikipedia.org/wiki/T-test#Dependent_t-test_for_paired_samples

**Examples**

```python
>>> from scipy import stats  # doctest: +SKIP
>>> np.random.seed(12345678)  # fix random seed to get same numbers  # doctest: +SKIP

>>> rvs1 = stats.norm.rvs(loc=5, scale=10, size=500)  # doctest: +SKIP
>>> rvs2 = (stats.norm.rvs(loc=5, scale=10, size=500) +  # doctest: +SKIP
...     stats.norm.rvs(scale=0.2, size=500))
>>> stats.ttest_rel(rvs1, rvs2)  # doctest: +SKIP
(0.24101764965300962, 0.80964043445811562)
```
dask array. stats.chisquare (f_obs, f_exp=None, ddof=0, axis=0)

Calculate 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 - 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

[1], [2]

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  # doctest: +SKIP
>>> chisquare([16, 18, 16, 14, 12, 12])  # doctest: +SKIP
(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])  # doctest: +SKIP
(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  # doctest: +SKIP
>>> obs.shape  # doctest: +SKIP
(6, 2)
>>> chisquare(obs)  # doctest: +SKIP
(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)  # doctest: +SKIP
(23.31034482758621, 0.015975692534127565)
>>> chisquare(obs.ravel())  # doctest: +SKIP
(23.31034482758621, 0.015975692534127565)

ddf is the change to make to the default degrees of freedom.

>>> chisquare([16, 18, 16, 14, 12, 12], ddof=1)  # doctest: +SKIP
(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])  # doctest: +SKIP
(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)  # doctest: +SKIP
(array([ 3.5 , 9.25]), array([ 0.62338763, 0.09949846]))

```
dask.array.stats.power_divergence(f_obs,f_exp=None,ddof=0,axis=0,lamba=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 \( \text{ddof} \) 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:

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;pearson&quot;</td>
<td>1</td>
<td>Pearson's chi-squared statistic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In this case, the function is equivalent to <code>stats.chisquare</code>.</td>
</tr>
<tr>
<td>&quot;log-likelihood&quot;</td>
<td>0</td>
<td>Log-likelihood ratio. Also known as the G-test [R5ed189a69e5c-3]_.</td>
</tr>
<tr>
<td>&quot;freeman-tukey&quot;</td>
<td>-1/2</td>
<td>Freeman-Tukey statistic.</td>
</tr>
<tr>
<td>&quot;mod-log-likelihood&quot;</td>
<td>-1</td>
<td>Modified log-likelihood ratio.</td>
</tr>
<tr>
<td>&quot;neyman&quot;</td>
<td>-2</td>
<td>Neyman's statistic.</td>
</tr>
<tr>
<td>&quot;cressie-read&quot;</td>
<td>2/3</td>
<td>The power recommended in [R5ed189a69e5c-5]_.</td>
</tr>
</tbody>
</table>

**Returns**

- **statistic** [float or ndarray] The Cressie-Read power divergence test statistic. The value is a float if \( \text{axis} \) 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 \( \text{ddof} \) and the return value \( \text{stat} \) 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

[1], [2], [3], [4], [5]

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):

```python
>>> 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:

```python
>>> 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.

```python
>>> obs = np.array([[16, 18, 16, 14, 12, 12],
                  [32, 24, 16, 28, 20, 24]]).T
>>> power_divergence(obs, lambda_="log-likelihood")
(array([ 2.00657316,  6.77634498]), array([ 0.84914504,  0.73575888,  0.5724067 ]))
```

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.

```python
>>> power_divergence(obs, axis=None)  # doctest: +SKIP
(23.31034482758621, 0.015975692534127565)
```

`ddof` is the change to make to the default degrees of freedom.

```python
>>> power_divergence([16, 18, 16, 14, 12, 12], ddof=1)  # doctest: +SKIP
(2.0, 0.73575888234288467)
```

The calculation of the p-values is done by broadcasting the test statistic with `ddof`.

```python
>>> power_divergence([16, 18, 16, 14, 12, 12], ddof=[0,1,2])  # doctest: +SKIP
(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:

```python
>>> power_divergence([16, 18, 16, 14, 12, 12], f_exp=[[16, 16, 16, 16, 16, 8]],
                    lambda_="log-likelihood")
(continues on next page)
```
... [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')

Compute the skewness of a data set.

For normally distributed data, the skewness should be about 0. For unimodal continuous distributions, a skewness value > 0 means that there is more weight in the right 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

[1]

Examples

```python
>>> from scipy.stats import skew  # doctest: +SKIP
>>> skew([1, 2, 3, 4, 5])  # doctest: +SKIP
0.0
>>> skew([2, 8, 0, 4, 1, 9, 9, 0])  # doctest: +SKIP
0.2650554122698573
```

dask.array.stats.skewtest (a, axis=0, nan_policy='propagate')

Test 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.

References

[1]

Examples

```python
>>> from scipy.stats import skewtest # doctest: +SKIP
>>> skewtest([1, 2, 3, 4, 5, 6, 7, 8]) # doctest: +SKIP
SkewtestResult(statistic=1.0108048609177787, pvalue=0.3121098361421897)
>>> skewtest([2, 8, 0, 4, 1, 9, 9, 0]) # doctest: +SKIP
SkewtestResult(statistic=0.44626385374196975, pvalue=0.6554066631275459)
>>> skewtest([1, 2, 3, 4, 5, 6, 7, 8000]) # doctest: +SKIP
SkewtestResult(statistic=3.571773510360407, pvalue=0.0003545719905823133)
>>> skewtest([100, 100, 100, 100, 100, 100, 100, 101]) # doctest: +SKIP
SkewtestResult(statistic=3.57177663478072, pvalue=0.000354567720281634)
```

dask.array.stats.kurtosis (a, axis=None, fisher=True, bias=True, nan_policy='propagate')

Compute 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

[1]

Examples

```python
>>> from scipy.stats import kurtosis # doctest: +SKIP
>>> kurtosis([1, 2, 3, 4, 5]) # doctest: +SKIP
-1.3
```

```python
dask.array.stats.kurtosistest(a, axis=0, nan_policy='propagate')
```

Test 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: \( \text{kurtosis} = \frac{3(n-1)}{n+1} \).

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 \( n>20 \). This function uses the method described in [1].

References

[1]

Examples

```python
>>> from scipy.stats import kurtosistest # doctest: +SKIP
>>> kurtosistest(list(range(20))) # doctest: +SKIP
KurtosistestResult(statistic=-1.7058104152122062, pvalue=0.08804338332528348)
```

```python
>>> np.random.seed(28041990) # doctest: +SKIP
>>> s = np.random.normal(0, 1, 1000) # doctest: +SKIP
>>> kurtosistest(s) # doctest: +SKIP
KurtosistestResult(statistic=1.2317590987707365, pvalue=0.21803908613450895)
```

```python
dask.array.stats.normaltest(a, axis=0, nan_policy='propagate')
```

Test whether a sample differs from a normal distribution.

---

4.7. Array 315
This function tests the null hypothesis that a sample comes from a normal distribution. It is based on D’Agostino and Pearson’s [1], [2] test that combines skew and kurtosis to produce an omnibus test of normality.

**Parameters**
- **a** [array_like] The array containing the sample 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.

**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**
[1], [2]

**Examples**

```python
>>> from scipy import stats # doctest: +SKIP
>>> pts = 1000  # doctest: +SKIP
>>> np.random.seed(28041990)  # doctest: +SKIP
>>> a = np.random.normal(0, 1, size=pts)  # doctest: +SKIP
>>> b = np.random.normal(2, 1, size=pts)  # doctest: +SKIP
>>> x = np.concatenate((a, b))  # doctest: +SKIP
>>> k2, p = stats.normaltest(x)  # doctest: +SKIP
>>> alpha = 1e-3  # doctest: +SKIP
>>> print("p = {0:.3g}".format(p))  # doctest: +SKIP
p = 3.27207e-11
>>> if p < alpha:  # null hypothesis: x comes from a normal distribution  
...    print("The null hypothesis can be rejected")  
... else:  
...    print("The null hypothesis cannot be rejected")
The null hypothesis can be rejected
```

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

[1], [2], [3]

Examples

```python
>>> import scipy.stats as stats  # doctest: +SKIP

[3] 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).

```python
>>> tillamook = [0.0571, 0.0813, 0.0831, 0.0976, 0.0817, 0.0859, 0.0735, ...
               0.0659, 0.0923, 0.0836]
>>> newport = [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]  # doctest: +SKIP
>>> stats.f_oneway(tillamook, newport, petersburg, magadan, tvarminne)  # doctest: +SKIP
(7.1210194716424473, 0.00028122423145345439)
```

**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 [1] for efficiency.

**References**

[1]

**Examples**

```python
>>> from scipy.stats import moment  # doctest: +SKIP
>>> moment([1, 2, 3, 4, 5], moment=1)  # doctest: +SKIP
0.0
>>> moment([1, 2, 3, 4, 5], moment=2)  # doctest: +SKIP
2.0
```

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
dask array.gufunc.apply_gufunc(func, signature, *args, **kwargs)

Apply a generalized ufunc or similar python function to arrays.

signature determines if the function consumes or produces core dimensions. The remaining dimensions in given input arrays (*args) are considered loop dimensions and are required to broadcast naturally against each other.

In other terms, this function is like np.vectorize, but for the blocks of dask arrays. If the function itself shall also be vectorized use vectorize=True for convenience.

Parameters

- **func** [callable] Function to call like func(*args, **kwargs) on input arrays (*args) that returns an array or tuple of arrays. If multiple arguments with non-matching dimensions are supplied, this function is expected to vectorize (broadcast) over axes of positional arguments in the style of NumPy universal functions [1] (if this is not the case, set vectorize=True). If this function returns multiple outputs, output_core_dims has to be set as well.

- **signature** string Specifies what core dimensions are consumed and produced by func. According to the specification of numpy.gufunc signature [2]

- ***args** [numeric] Input arrays or scalars to the callable function.

- **axes** List of tuples, optional, keyword only A list of tuples with indices of axes a generalized ufunc should operate on. For instance, for a signature of "(i,j),(j,k)->(i,k)" appropriate for matrix multiplication, the base elements are two-dimensional matrices and these are taken to be stored in the two last axes of each argument. The corresponding axes keyword would be [(-2, -1), (-2, -1), (-2, -1)]. For simplicity, for generalized ufuncs that operate on 1-dimensional arrays (vectors), a single integer is accepted instead of a single-element tuple, and for generalized ufuncs for which all outputs are scalars, the output tuples can be omitted.

- **axis** int, optional, keyword only A single axis over which a generalized ufunc should operate. This is a short-cut for ufuncs that operate over a single, shared core dimension, equivalent to passing in axes with entries of (axis,) for each single-core-dimension argument and () for all others. For instance, for a signature "(i),(i)->()", it is equivalent to passing in axes=[(axis,), (axis,), ()].

- **keepdims** bool, optional, keyword only If this is set to True, axes which are reduced over will be left in the result as a dimension with size one, so that the result will broadcast correctly against the inputs. This option can only be used for generalized ufuncs that operate on inputs that all have the same number of core dimensions and with outputs that have no core dimensions, i.e., with signatures like "(i),(i)->()" or "(m,m)->()". If used, the location of the dimensions in the output can be controlled with axes and axis.

- **output_dtypes** [Optional, dtype or list of dtypes, keyword only] Valid numpy dtype specification or list thereof. If not given, a call of func with a small set of data is performed in order to try to automatically determine the output dtypes.

- **output_sizes** [dict, optional, keyword only] Optional mapping from dimension names to sizes for outputs. Only used if new core dimensions (not found on inputs) appear on outputs.
vectorize: bool, keyword only If set to True, np.vectorize is applied to func for convenience. Defaults to False.

allow_rechunk: Optional, bool, keyword only Allows rechunking, otherwise chunk sizes need to match and core dimensions are to consist only of one chunk. Warning: enabling this can increase memory usage significantly. Defaults to False.

**kwargs [dict] Extra keyword arguments to pass to func

Returns

Single dask.array.Array or tuple of dask.array.Array

References

[1], [2]

Examples

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

>>> def stats(x):
...     return np.mean(x, axis=-1), np.std(x, axis=-1)

>>> a = da.random.normal(size=(10, 20, 30), chunks=(5, 10, 30))
>>> mean, std = da.apply_gufunc(stats, "(i)->(),()", a)
>>> mean.compute().shape
(10, 20)

>>> def outer_product(x, y):
...     return np.einsum("i,j->ij", x, y)

>>> a = da.random.normal(size=(20, 30), chunks=(10, 30))
>>> b = da.random.normal(size=(10, 1, 40), chunks=(5, 1, 40))
>>> c = da.apply_gufunc(outer_product, "(i),(j)->(i,j)", a, b, vectorize=True)
>>> c.compute().shape
(10, 20, 30, 40)
```

dask.array.gufunc.as_gufunc(signature=None, **kwargs)

Decorator for dask.array.gufunc.

Parameters

signature [String] Specifies what core dimensions are consumed and produced by func. According to the specification of numpy.gufunc signature [2]

axes: List of tuples, optional, keyword only A list of tuples with indices of axes a generalized ufunc should operate on. For instance, for a signature of "(i, j), (j, k) -> (i, k)" appropriate for matrix multiplication, the base elements are two-dimensional matrices and these are taken to be stored in the two last axes of each argument. The corresponding axes keyword would be [(-2, -1), (-2, -1), (-2, -1)]. For simplicity, for generalized ufuncs that operate on 1-dimensional arrays (vectors), a single integer is accepted instead of a single-element tuple, and for generalized ufuncs for which all outputs are scalars, the output tuples can be omitted.

axis: int, optional, keyword only A single axis over which a generalized ufunc should operate. This is a short-cut for ufuncs that operate over a single, shared core dimension, equivalent to passing in axes with entries of (axis,) for each single-core-dimension argument and ()
for all others. For instance, for a signature "(i),(i)->()", it is equivalent to passing in
axes=[(axis,), (axis,), ()].

keepdims: bool, optional, keyword only  If this is set to True, axes which are reduced over will
be left in the result as a dimension with size one, so that the result will broadcast correctly
against the inputs. This option can only be used for generalized ufuncs that operate on
inputs that all have the same number of core dimensions and with outputs that have no core
dimensions, i.e., with signatures like "(i),(i)->()" or "(m,m)->()". If used, the
location of the dimensions in the output can be controlled with axes and axis.

output_dtypes [Optional, dtype or list of dtypes, keyword only] Valid numpy dtype specification
or list thereof. If not given, a call of func with a small set of data is performed in order
to try to automatically determine the output dtypes.

output_sizes [dict, optional, keyword only] Optional mapping from dimension names to sizes
for outputs. Only used if new core dimensions (not found on inputs) appear on outputs.

vectorize: bool, keyword only If set to True, np.vectorize is applied to func for con-
venience. Defaults to False.

allow_rechunk: Optional, bool, keyword only Allows rechunking, otherwise chunk sizes
need to match and core dimensions are to consist only of one chunk. Warning: enabling
this can increase memory usage significantly. Defaults to False.

Returns

Decorator for 'pyfunc' that itself returns a 'gufunc'.

References

[1], [2]

Examples

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

>>> a = da.random.normal(size=(10, 20, 30), chunks=(5, 10, 30))
>>> @da.as_gufunc("(i)->(),()", output_dtypes=(float, float))
... def stats(x):
...     return np.mean(x, axis=-1), np.std(x, axis=-1)
>>> mean, std = stats(a)
>>> mean.compute().shape
(10, 20)

>>> a = da.random.normal(size=(20, 30), chunks=(10, 30))
>>> b = da.random.normal(size=(10, 1, 40), chunks=(5, 1, 40))
>>> @da.as_gufunc("(i),(j)->(i,j)", output_dtypes=float, vectorize=True)
... def outer_product(x, y):
...     return np.einsum("i,j->ij", x, y)
>>> c = outer_product(a, b)
>>> c.compute().shape
(10, 20, 30, 40)
```

dask.array.gufunc.gufunc(pyfunc, **kwars)

Binds pyfunc into dask.array.apply_gufunc when called.

Parameters
pyfunc  [callable] Function to call like \texttt{func(*args, **kwargs)} on input arrays \((\ast \texttt{args})\) that returns an array or tuple of arrays. If multiple arguments with non-matching dimensions are supplied, this function is expected to vectorize (broadcast) over axes of positional arguments in the style of NumPy universal functions \cite{1} (if this is not the case, set \texttt{vectorize=True}). If this function returns multiple outputs, \texttt{output_core_dims} has to be set as well.

signature  [String, keyword only] Specifies what core dimensions are consumed and produced by \texttt{func}. According to the specification of numpy.gufunc signature \cite{2}

axes:  List of tuples, optional, keyword only A list of tuples with indices of axes a generalized ufunc should operate on. For instance, for a signature of "\((i, j), (j, k) -> (i, k)\)" appropriate for matrix multiplication, the base elements are two-dimensional matrices and these are taken to be stored in the two last axes of each argument. The corresponding axes keyword would be \([(-2, -1), (-2, -1), (-2, -1)]\). For simplicity, for generalized ufuncs that operate on 1-dimensional arrays (vectors), a single integer is accepted instead of a single-element tuple, and for generalized ufuncs for which all outputs are scalars, the output tuples can be omitted.

axis:  int, optional, keyword only A single axis over which a generalized ufunc should operate. This is a short-cut for ufuncs that operate over a single, shared core dimension, equivalent to passing in axes with entries of \((\texttt{axis,})\) for each single-core-dimension argument and \(()\) for all others. For instance, for a signature "\((i), (i) -> ()\)", it is equivalent to passing in \texttt{axes=[(axis, ), (axis, ), ()]}

keepdims:  bool, optional, keyword only If this is set to \texttt{True}, axes which are reduced over will be left in the result as a dimension with size one, so that the result will broadcast correctly against the inputs. This option can only be used for generalized ufuncs that operate on inputs that all have the same number of core dimensions and with outputs that have no core dimensions, i.e., with signatures like "\((i), (i) -> ()\)" or "\((m,m) -> ()\)". If used, the location of the dimensions in the output can be controlled with axes and axis.

output_dtypes  [Optional, dtype or list of dtypes, keyword only] Valid numpy dtype specification or list thereof. If not given, a call of \texttt{func} with a small set of data is performed in order to try to automatically determine the output dtypes.

output_sizes  [dict, optional, keyword only] Optional mapping from dimension names to sizes for outputs. Only used if new core dimensions (not found on inputs) appear on outputs.

vectorize:  bool, keyword only If set to \texttt{True}, \texttt{np.vectorize} is applied to \texttt{func} for convenience. Defaults to \texttt{False}.

allow_rechunk:  Optional, bool, keyword only Allows rechunking, otherwise chunk sizes need to match and core dimensions are to consist only of one chunk. Warning: enabling this can increase memory usage significantly. Defaults to \texttt{False}.

Returns

Wrapped function

References

\cite{1}, \cite{2}
dask Documentation, Release 1.2.2

Examples
>>> import dask.array as da
>>> import numpy as np
>>> a = da.random.normal(size=(10,20,30), chunks=(5, 10, 30))
>>> def stats(x):
...
return np.mean(x, axis=-1), np.std(x, axis=-1)
>>> gustats = da.gufunc(stats, signature="(i)->(),()", output_dtypes=(float,
˓→float))
>>> mean, std = gustats(a)
>>> mean.compute().shape
(10, 20)
>>> a = da.random.normal(size=(
20,30), chunks=(10, 30))
>>> b = da.random.normal(size=(10, 1,40), chunks=(5, 1, 40))
>>> def outer_product(x, y):
...
return np.einsum("i,j->ij", x, y)
>>> guouter_product = da.gufunc(outer_product, signature="(i),(j)->(i,j)", output_
˓→dtypes=float, vectorize=True)
>>> c = guouter_product(a, b)
>>> c.compute().shape
(10, 20, 30, 40)

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 other objects]
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)

4.7. Array

323


```python
>>> 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.

```python
>>> d = da.arange(5, chunks=2)
>>> e = da.arange(5, chunks=2)
```

```python
>>> 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.

```python
>>> 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.

```python
>>> 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.

```python
>>> b = a.map_blocks(lambda x: x[None, :, None], chunks=(1, 6, 1),
                   new_axis=[0, 2])
```

If `chunks` is specified but `new_axis` is not, then it is inferred to add the necessary number of axes on the left.

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.

```python
>>> x = da.arange(1000, chunks=(100,))
>>> y = da.arange(100, chunks=(10,))
```

The relevant attribute to match is `numblocks`.

```python
>>> x.numblocks
(10,)
>>> y.numblocks
(10,)
```

If these match (up to broadcasting rules) then we can map arbitrary functions across blocks

```python
>>> def func(a, b):
...     return np.array([a.max(), b.max()])
```

```python
>>> da.map_blocks(func, x, y, chunks=(2,), dtype='i8')
dask.array<func, shape=(20,), dtype=int64, chunksize=(2,)>
```

```python
>>> _.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 get information about where it is in the array by accepting a special `block_info` keyword argument.
>>> def func(block, block_info=None):
...    pass

This will receive the following information:

>>> block_info # doctest: +SKIP
{0: {'shape': (1000,),
     'num-chunks': (10,),
     'chunk-location': (4,),
     'array-location': [(400, 500)],
None: {'shape': (1000,),
     'num-chunks': (10,),
     'chunk-location': (4,),
     'array-location': [(400, 500)],
     'chunk-shape': (100,),
     'dtype': dtype('float64')}}

For each argument and keyword arguments that are dask arrays (the positions of which are the first index), you will receive the shape of the full array, the number of chunks of the full array in each dimension, the chunk location (for example the fourth chunk over in the first dimension), and the array location (for example the slice corresponding to 40:50). The same information is provided for the output, with the key None, plus the shape and dtype that should be returned.

These features can be combined to synthesize an array from scratch, for example:

>>> def func(block_info=None):
...    loc = block_info[None]['array-location'][0]
...    return np.arange(loc[0], loc[1])

>>> da.map_blocks(func, chunks=((4, 4),), dtype=np.float_)
dask.array<func, shape=(8,), dtype=float64, chunksize=(4,)>

>>> _.compute()
array([0, 1, 2, 3, 4, 5, 6, 7])

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, name='increment') # doctest: +SKIP
dask.array<increment, shape=(100,), dtype=int64, chunksize=(10,)>

dask.array.core.blockwise(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 blockwise 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 blockwise 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

### Examples

2D embarrassingly parallel operation from two arrays, \(x\), and \(y\).

```python
>>> z = blockwise(operator.add, 'ij', x, 'ij', y, 'ij', dtype='f8')  # z = x + y
```

Outer product multiplying \(x\) by \(y\), two 1-d vectors

```python
>>> z = blockwise(operator.mul, 'ij', x, 'i', y, 'j', dtype='f8')  # doctest: +SKIP
z = x.T
```

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 \(ij \rightarrow ji\).

We can compose these same patterns with more variables and more complex in-memory functions

\[z = X + Y .T\]

```python
>>> def f(x):
...     return x[:, None] * np.ones((1, 5))
```

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

```python
>>> def sequence_dot(x_blocks, y_blocks):
...     result = 0
...     for x, y in zip(x_blocks, y_blocks):
...         result += x.dot(y)
...     return result
```

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.

```python
>>> def f(x):
...     return x[:, None] * np.ones((1, 5))
```
New dimensions can also be multi-chunk by specifying a tuple of chunk sizes. This has limited utility as is (because the chunks are all the same), but the resulting graph can be modified to achieve more useful results (see `da.map_blocks`).

```python
>>> z = blockwise(f, 'az', x, 'a', new_axes={'z': (5, 5)}, dtype=x.dtype)  # doctest: +SKIP
```

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.

```python
>>> def double(x):
...     return np.concatenate([x, x])
...
```

```python
>>> y = blockwise(double, 'ij', x, 'ij', 
...    adjust_chunks={'i': lambda n: 2 * n}, dtype=x.dtype)  # doctest: +SKIP
```

Include literals by indexing with None

```python
>>> y = blockwise(add, 'ij', x, 'ij', 1234, None, dtype=x.dtype)  # doctest: +SKIP
```

dask.array.core.normalize_chunks(chunks, shape=None, limit=None, dtype=None, previous_chunks=None)

Normalize chunks to tuple of tuples

This takes in a variety of input types and information and produces a full tuple-of-tuples result for chunks, suitable to be passed to Array or rechunk or any other operation that creates a Dask array.

Parameters

- **chunks**: `tuple, int, dict, or string` The chunks to be normalized. See examples below for more details
- **shape**: `Tuple[int]` The shape of the array
- **limit**: `int (optional)` The maximum block size to target in bytes, if freedom is given to choose
- **dtype**: `np.dtype`
- **previous_chunks**: `Tuple[Tuple[int]] optional` Chunks from a previous array that we should use for inspiration when rechunking auto dimensions. If not provided but auto-chunking exists then auto-dimensions will prefer square-like chunk shapes.

Examples

Specify uniform chunk sizes

```python
>>> normalize_chunks((2, 2), shape=(5, 6))
((2, 2, 1), (2, 2, 2))
```

Also passes through fully explicit tuple-of-tuples

```python
>>> normalize_chunks(((2, 2, 1), (2, 2, 2)), shape=(5, 6))
((2, 2, 1), (2, 2, 2))
```
Cleans up lists to tuples

```python
>>> normalize_chunks([[2, 2], [3, 3]])
((2, 2), (3, 3))
```

Expands integer inputs 10 -> (10, 10)

```python
>>> normalize_chunks(10, shape=(30, 5))
((10, 10, 10), (5,))
```

Expands dict inputs

```python
>>> normalize_chunks({0: 2, 1: 3}, shape=(6, 6))
((2, 2, 2), (3, 3))
```

The values -1 and None get mapped to full size

```python
>>> normalize_chunks((5, -1), shape=(10, 10))
((5, 5), (10,))
```

Use the value “auto” to automatically determine chunk sizes along certain dimensions. This uses the `limit=` and `dtype=` keywords to determine how large to make the chunks. The term “auto” can be used anywhere an integer can be used. See array chunking documentation for more information.

```python
>>> normalize_chunks("auto", shape=(20,), limit=5, dtype='uint8')
((5, 5, 5, 5),)
```

You can also use byte sizes (see `dask.utils.parse_bytes`) in place of “auto” to ask for a particular size

```python
>>> normalize_chunks("1kiB", shape=(2000,), dtype='float32')
((250, 250, 250, 250, 250, 250, 250, 250),)
```

Respects null dimensions

```python
>>> normalize_chunks((), shape=(0, 0))
((0,), (0,))
```

Array Methods

```python
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)
This docstring was copied from numpy.ndarray.all.
Some inconsistencies with the Dask version may exist.
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)
This docstring was copied from numpy.ndarray.any.
Some inconsistencies with the Dask version may exist.
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)
This docstring was copied from numpy.ndarray.argmax.
Some inconsistencies with the Dask version may exist.
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)
This docstring was copied from numpy.ndarray.argmin.
Some inconsistencies with the Dask version may exist.
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

argtopk (k, axis=-1, split_every=None)
The indices of the top k elements of an array.
See da.argtopk for docstring

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.

- ‘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.

blocks
Slice an array by blocks

This allows blockwise slicing of a Dask array. You can perform normal Numpy-style slicing but now rather than slice elements of the array you slice along blocks so, for example, x.blocks[0, ::2] produces a new dask array with every other block in the first row of blocks.

You can index blocks in any way that could index a numpy array of shape equal to the number of blocks in each dimension, (available as array.numblocks). The dimension of the output array will be the same as the dimension of this array, even if integer indices are passed. This does not support slicing with np.newaxis or multiple lists.

Returns
A Dask array

Examples

```python
>>> import dask.array as da
>>> x = da.arange(10, chunks=2)
>>> x.blocks[0].compute()
array([0, 1])
>>> x.blocks[:3].compute()
array([0, 1, 2, 3, 4, 5])
>>> x.blocks[::2].compute()
array([0, 1, 4, 5, 8, 9])
>>> x.blocks[[-1, 0]].compute()
array([8, 9, 0, 1])
```

choose (choices, out=None, mode=’raise’)

This docstring was copied from numpy.ndarray.choose.

Some inconsistencies with the Dask version may exist.

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 \((\text{min} = \text{None}, \text{max} = \text{None}, \text{out} = \text{None})\)

This docstring was copied from numpy.ndarray.clip.

Some inconsistencies with the Dask version may exist.

Return an array whose values are limited to \([\text{min}, \text{max}]\). One of \text{max} or \text{min} must be given.

Refer to \text{numpy.clip} for full documentation.

See also:

\text{numpy.clip} equivalent function

copy()

Copy array. This is a no-op for dask.arrays, which are immutable

cumprod \((\text{axis} = \text{None}, \text{dtype} = \text{None}, \text{out} = \text{None})\)

See da.cumprod for docstring

cumsum \((\text{axis} = \text{None}, \text{dtype} = \text{None}, \text{out} = \text{None})\)

See da.cumsum for docstring

dot \((b, \text{out} = \text{None})\)

This docstring was copied from numpy.ndarray.dot.

Some inconsistencies with the Dask version may exist.

Dot product of two arrays.

Refer to \text{numpy.dot} for full documentation.

See also:

\text{numpy.dot} equivalent function

Examples

```python
>>> a = np.eye(2)  # doctest: +SKIP
>>> b = np.ones((2, 2)) * 2  # doctest: +SKIP
>>> a.dot(b)  # doctest: +SKIP
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```python
>>> a.dot(b).dot(b)  # doctest: +SKIP
array([[ 8.,  8.],
       [ 8.,  8.]])
```

flatten \((\text{order})\)

This docstring was copied from numpy.ndarray.ravel.

Some inconsistencies with the Dask version may exist.

Return a flattened array.

Refer to \text{numpy.ravel} for full documentation.

See also:

\text{numpy.ravel} equivalent function
**ndarray.flat** a flat iterator on the array.

**itemsize**
Length of one array element in bytes

**map_blocks** *(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 other objects]
- **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**

```python
>>> import dask.array as da
>>> x = da.arange(6, chunks=3)

>>> x.map_blocks(lambda x: x * 2).compute()
darray([ 0,  2,  4,  6,  8, 10])
```

The `da.map_blocks` function can also accept multiple arrays.

```python
>>> 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()
darray([ 0,  2,  6, 12, 20])
```

If the function changes shape of the blocks then you must provide `chunks` explicitly.

```python
>>> 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.

```python
>>> 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.

```python
>>> b = a.map_blocks(lambda x: x[None, :, None], chunks=(1, 6, 1),
                   new_axis=[0, 2])
```

If `chunks` is specified but `new_axis` is not, then it is inferred to add the necessary number of axes on the left.

`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.

```python
>>> x = da.arange(1000, chunks=(100,))
>>> y = da.arange(100, chunks=(10,))
```

The relevant attribute to match is numblocks.

```python
>>> x.numblocks
(10,)
>>> y.numblocks
(10,)
```

If these match (up to broadcasting rules) then we can map arbitrary functions across blocks

```python
>>> def func(a, b):
...    return np.array([a.max(), b.max()])
```

```python
>>> da.map_blocks(func, x, y, chunks=(2,), dtype='i8')
dask.array<func, shape=(20,), dtype=int64, chunksize=(2,)>
```

```python
>>> _.compute()
```

Your block function get information about where it is in the array by accepting a special `block_info` keyword argument.

```python
>>> def func(block, block_info=None):
...    pass
```

This will receive the following information:

```python
>>> block_info  # doctest: +SKIP
(0: {'shape': (1000,),
     'num-chunks': (10,),
     'chunk-location': (4,),
     'array-location': [(400, 500)]},
  None: {'shape': (1000,),
          'num-chunks': (10,),
          'chunk-location': (4,),
          'array-location': [(400, 500)],
          'chunk-shape': (100,),
          'dtype': dtype('float64'))
```
For each argument and keyword arguments that are dask arrays (the positions of which are the first index),
you will receive the shape of the full array, the number of chunks of the full array in each dimension, the
chunk location (for example the fourth chunk over in the first dimension), and the array location (for
example the slice corresponding to 40:50). The same information is provided for the output, with the
key None, plus the shape and dtype that should be returned.

These features can be combined to synthesize an array from scratch, for example:

>>> def func(block_info=None):
...     loc = block_info[None]['array-location'][0]
...     return np.arange(loc[0], loc[1])

>>> da.map_blocks(func, chunks=((4, 4),), dtype=np.float_)
dask.array<func, shape=(8,), dtype=float64, chunksize=(4,)>

>>> _.compute()
array([0, 1, 2, 3, 4, 5, 6, 7])

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, name='increment')
# doctest: +SKIP
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 neigh-
boring strips.

Parameters

func: function The function to apply to each extended block

depth: int, tuple, or dict The number of elements that each block should share with its
neighbors. If a tuple or dict then this can be different per axis

boundary: str, tuple, dict How to handle the boundaries. Values include ‘reflect’, ‘peri-
dodic’, ‘nearest’, ‘none’, or any constant value like 0 or np.nan

trim: bool Whether or not to trim depth elements from each block after calling the map
function. Set this to False if your mapping function already 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]])

```python
>>> func = lambda x: x + x.size
>>> depth = {0: 1, 1: 1}
>>> boundary = {0: 'reflect', 1: 'none'}
>>> d.map_overlap(func, depth, boundary).compute()  # doctest: +NORMALIZE_WHITESPACE
array([[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23],
[24, 25, 26, 27]])
```

max (axis=None, out=None, keepdims=False)
This docstring was copied from numpy.ndarray.max.
Some inconsistencies with the Dask version may exist.
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)
This docstring was copied from numpy.ndarray.mean.
Some inconsistencies with the Dask version may exist.
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)
This docstring was copied from numpy.ndarray.min.
Some inconsistencies with the Dask version may exist.
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 nth 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.

ddf  [int, optional]  “Delta Degrees of Freedom”: the divisor used in the calculation is \( N - ddf \), where \( N \) represents the number of elements. By default ddf is zero.

Returns

moment  [ndarray]

References

Computation of Covariances and Arbitrary-Order Statistical Moments” (PDF), Technical Report SAND2008-6212, Sandia National Laboratories

[1]
Refer to `numpy.ravel` for full documentation.

**See also:**

- `numpy.ravel` equivalent function
- `ndarray.flat` a flat iterator on the array.

### rechunk

```python
rechunk(chunks, threshold=None, block_size_limit=None)
```

See `da.rechunk` for docstring

### repeat

```python
repeat(repeats, axis=None)
```

This docstring was copied from `numpy.ndarray.repeat`.

Some inconsistencies with the Dask version may exist.

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See also:**

- `numpy.repeat` equivalent function

### reshape

```python
reshape(shape, order='C')
```

This docstring was copied from `numpy.ndarray.reshape`.

Some inconsistencies with the Dask version may exist.

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

**See also:**

- `numpy.reshape` equivalent function

### Notes

Unlike the free function `numpy.reshape`, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

### round

```python
round(decimals=0, out=None)
```

This docstring was copied from `numpy.ndarray.round`.

Some inconsistencies with the Dask version may exist.

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** (<code>axis=None</code>)

This docstring was copied from numpy.ndarray.squeeze.
Some inconsistencies with the Dask version may exist.
Remove single-dimensional entries from the shape of <code>a</code>. Refer to <code>numpy.squeeze</code> for full documentation.

See also:

<code>numpy.squeeze</code> equivalent function

**std** (<code>axis=None, dtype=None, out=None, ddof=0, keepdims=False</code>)

This docstring was copied from numpy.ndarray.std.
Some inconsistencies with the Dask version may exist.
Returns the standard deviation of the array elements along given axis.
Refer to <code>numpy.std</code> for full documentation.

See also:

<code>numpy.std</code> equivalent function

**store** (<code>targets, lock=True, regions=None, compute=True, return_stored=False, **kwargs</code>)

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 <code>np.array(myarray)</code> instead.

Parameters

**sources**: Array or iterable of Arrays

**targets**: array-like or Delayed or iterable of array-likes and/or Delayeds

These should support setitem syntax <code>target[10:20] = ...</code>

**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 list of tuples of slices

Each region tuple in <code>regions</code> should be such that <code>target[region].shape = source.shape</code> for the corresponding source and target in sources and targets, respectively. If this is a tuple, the contents will be assumed to be slices, so do not provide a tuple of tuples.

**compute**: boolean, optional

If true compute immediately, return <code>dask.delayed.</code> Delayed otherwise

**return_stored**: boolean, optional

Optionally return the stored result (default False).

Examples

```python
>>> x = ...  # doctest: +SKIP
```
```python
>>> import h5py  # doctest: +SKIP
>>> f = h5py.File('myfile.hdf5')  # doctest: +SKIP
>>> dset = f.create_dataset('/data', shape=x.shape,
...     chunks=x.chunks,
...     dtype='f8')  # doctest: +SKIP
```
Parameters

**optimize_graph** [bool, optional] If True [default], the graph is optimized before converting into dask.delayed objects.

See also:

dask.array.from_delayed
dask.array.to_hdf5 *(filename, datapath, **kwargs)*
Store array in HDF5 file

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

Optionally provide arguments as though to h5py.File.create_dataset

```python
>>> x.to_hdf5('myfile.hdf5', '/x', compression='lzf', shuffle=True)  # doctest: +SKIP
```

See also:

dak.store, h5py.File.create_dataset
to_tiledb *(uri, *args, **kwargs)*
Save array to the TileDB storage manager

See function to_tiledb() for argument documentation.
See https://docs.tiledb.io for details about the format and engine.
to_zarr *(*args, **kwargs)*
Save array to the zarr storage format

See https://zarr.readthedocs.io for details about the format.
See function to_zarr() for parameters.

topk *(k, axis=-1, split_every=None)*
The top k elements of an array.

See da.topk for docstring
trace *(offset=0, axis1=0, axis2=1, dtype=None, out=None)*
This docstring was copied from numpy.ndarray.trace.

Some inconsistencies with the Dask version may exist.
Return the sum along diagonals of the array.
Refer to numpy.trace for full documentation.

See also:

numpy.trace equivalent function

transpose *(*axes)*
This docstring was copied from numpy.ndarray.transpose.

Some inconsistencies with the Dask version may exist.
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 \( a.shape = (i[0], i[1], \ldots, i[n-2], i[n-1]) \), then \( a.transpose().shape = (i[n-1], i[n-2], \ldots, i[1], i[0]) \).

### 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

```python
>>> a = np.array([[1, 2], [3, 4]])  # doctest: +SKIP
>>> a  # doctest: +SKIP
array([[1, 2],
       [3, 4]])
>>> a.transpose()  # doctest: +SKIP
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))  # doctest: +SKIP
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)  # doctest: +SKIP
array([[1, 3],
       [2, 4]])
```

### var

\( (axis=None, dtype=None, out=None, ddof=0, keepdims=False) \)

This docstring was copied from numpy.ndarray.var.

Some inconsistencies with the Dask version may exist.

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:

```python
>>> 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

4.7.2 Best Practices

It is easy to get started with Dask arrays, but using them well does require some experience. This page contains suggestions for best practices, and includes solutions to common problems.

Use NumPy

If your data fits comfortably in RAM and you are not performance bound, then using NumPy might be the right choice. Dask adds another layer of complexity which may get in the way.

If you are just looking for speedups rather than scalability then you may want to consider a project like Numba

Select a good chunk size

A common performance problem among Dask Array users is that they have chosen a chunk size that is either too small (leading to lots of overhead) or poorly aligned with their data (leading to inefficient reading).

While optimal sizes and shapes are highly problem specific, it is rare to see chunk sizes below 100 MB in size. If you are dealing with float64 data then this is around (4000, 4000) in size for a 2D array or (100, 400, 400) for a 3D array.

You want to choose a chunk size that is large in order to reduce the number of chunks that Dask has to think about (which affects overhead) but also small enough so that many of them can fit in memory at once. Dask will often have as many chunks in memory as twice the number of active threads.
Orient your chunks

When reading data you should align your chunks with your storage format. Most array storage formats store data in chunks themselves. If your Dask array chunks aren’t multiples of these chunk shapes then you will have to read the same data repeatedly, which can be expensive. Note though that often storage formats choose chunk sizes that are much smaller than is ideal for Dask, closer to 1MB than 100MB. In these cases you should choose a Dask chunk size that aligns with the storage chunk size and that every Dask chunk dimension is a multiple of the storage chunk dimension.

So for example if we have an HDF file that has chunks of size (128, 64), we might choose a chunk shape of (1280, 6400).

```python
>>> import h5py

>>> storage = h5py.File('myfile.hdf5')['x']

>>> storage.chunks
(128, 64)

>>> import dask.array as da

>>> x = da.from_array(storage, chunks=(1280, 6400))
```

Note that if you provide `chunks='auto'` then Dask Array will look for a `.chunks` attribute and use that to provide a good chunking.

Avoid Oversubscribing Threads

By default Dask will run as many concurrent tasks as you have logical cores. It assumes that each task will consume about one core. However, many array-computing libraries are themselves multi-threaded, which can cause contention and low performance. In particular the BLAS/LAPACK libraries that back most of NumPy’s linear algebra routines are often multi-threaded, and need to be told to use only one thread explicitly. You can do this with the following environment variables (using bash `export` command below, but this may vary depending on your operating system).

```bash
export OMP_NUM_THREADS=1
export MKL_NUM_THREADS=1
export OPENBLAS_NUM_THREADS=1
```

You need to run this before you start your Python process for it to take effect.

Consider Xarray

The Xarray package wraps around Dask Array, and so offers the same scalability, but also adds convenience when dealing with complex datasets. In particular Xarray can help with the following:

1. Manage multiple arrays together as a consistent dataset
2. Read from a stack of HDF or NetCDF files at once
3. Switch between Dask Array and NumPy with a consistent API

Xarray is used in wide range of fields, including physics, astronomy, geoscience, microscopy, bioinformatics, engineering, finance, and deep learning. Xarray also has a thriving user community that is good at providing support.

Build your own Operations

Often we want to perform computations for which there is no exact function in Dask Array. In these cases we may be able to use some of the more generic functions to build our own. These include:
These functions may help you to apply a function that you write for NumPy functions onto larger Dask arrays.

### 4.7.3 Chunks

Dask arrays are composed of many NumPy arrays. How these arrays are arranged can significantly affect performance. For example, for a square array you might arrange your chunks along rows, along columns, or in a more square-like fashion. Different arrangements of NumPy arrays will be faster or slower for different algorithms.

Thinking about and controlling chunking is important to optimize advanced algorithms.

#### Specifying Chunk shapes

We always specify a `chunks` argument to tell `dask.array` how to break up the underlying array into chunks. We can specify `chunks` in a variety of ways:

1. A uniform dimension size like `1000`, meaning chunks of size `1000` in each dimension
2. A uniform chunk shape like `(1000, 2000, 3000)`, meaning chunks of size `1000` in the first axis, `2000` in the second axis, and `3000` in the third
3. Fully explicit sizes of all blocks along all dimensions, like `{(1000, 1000, 500), (400, 400), (5, 5, 5, 5, 5)}`
4. A dictionary specifying chunk size per dimension like `{0: 1000, 1: 2000, 2: 3000}`. This is just another way of writing the forms 2 and 3 above

Your chunks input will be normalized and stored in the third and most explicit form. Note that `chunks` stands for “chunk shape” rather than “number of chunks”, so specifying `chunks=1` means that you will have many chunks, each with exactly one element.

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. Chunk sizes between 10MB-1GB are common, depending on the availability of RAM and the duration of computations
4. 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
5. Chunks should align with your storage, if applicable.

Array data formats are often chunked as well. When loading or saving data, if is useful to have Dask array chunks that are aligned with the chunking of your storage, often an even multiple times larger in each direction.
Unknown Chunks

Some arrays have unknown chunk sizes. This arises whenever the size of an array depends on lazy computations that
we haven’t yet performed like the following:

```python
x = x[x > 100]  # don't know how many values are greater than 100 ahead of time
```

Operations like the above result in arrays with unknown shapes and unknown chunk sizes. Unknown values within
shape or chunks are designated using `np.nan` rather than an integer. These arrays support many (but not all) opera-
tions. In particular, operations like slicing are not possible and will result in an error.

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

>>> x[100]
ValueError: Array chunk sizes unknown
```

This can also happen when creating a Dask array from a Dask DataFrame:

```python
>>> ddf = dask.dataframe.from_pandas(...) >>> ddf.to_dask_array...
... dask.array<values, shape=(nan, 2), dtype=float64, chunksize=(nan, 2)>
```

For details on how to avoid unknown chunk sizes, look at how to create a Dask array from a Dask DataFrame in the
documentation on Dask array creation.

Chunks Examples

In this example we show how different inputs for `chunks=` cut up the following array:

```plaintext
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
```

Here, we show how different `chunks=` arguments split the array into different blocks

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

```plaintext
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:

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

(continues on next page)
chunks=(3, 2): Asymmetric but repeated blocks of size (3, 2):

<table>
<thead>
<tr>
<th>1 2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 8 9 0 1 2</td>
</tr>
<tr>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>9 0 1 2 3 4</td>
</tr>
<tr>
<td>5 6 7 8 9 0</td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>1 2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 8 9 0 1 2</td>
</tr>
<tr>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>9 0 1 2 3 4</td>
</tr>
<tr>
<td>5 6 7 8 9 0</td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>1 2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 8 9 0 1 2</td>
</tr>
<tr>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>9 0 1 2 3 4</td>
</tr>
<tr>
<td>5 6 7 8 9 0</td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>1 2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 8 9 0 1 2</td>
</tr>
<tr>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>9 0 1 2 3 4</td>
</tr>
<tr>
<td>5 6 7 8 9 0</td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
</tr>
</tbody>
</table>

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.
Loading Chunked Data

Modern NDArray storage formats like HDF5, NetCDF, TIFF, and Zarr, allow arrays to be stored in chunks or tiles so that blocks of data can be pulled out efficiently without having to seek through a linear data stream. It is best to align the chunks of your Dask array with the chunks of your underlying data store.

However, data stores often chunk more finely than is ideal for Dask array, so it is common to choose a chunking that is a multiple of your storage chunk size, otherwise you might incur high overhead.

For example, if you are loading a data store that is chunked in blocks of \((100, 100)\), then you might choose a chunking more like \((1000, 2000)\) that is larger, but still evenly divisible by \((100, 100)\). Data storage technologies will be able to tell you how their data is chunked.

Rechunking

\[ \text{rechunk}(x, \text{chunks}[, \text{threshold, block_size_limit}]) \] Convert blocks in dask array \(x\) for new chunks.

Sometimes you need to change the chunking layout of your data. For example, perhaps it comes to you chunked row-wise, but you need to do an operation that is much faster if done across columns. You can change the chunking with the \text{rechunk} method.

\[ x = x.\text{rechunk}((50, 1000)) \]

Rechunking across axes can be expensive and incur a lot of communication, but Dask array has fairly efficient algorithms to accomplish this.

You can pass rechunk any valid chunking form:

\[ x = x.\text{rechunk}(1000) \]
\[ x = x.\text{rechunk}((50, 1000)) \]
\[ x = x.\text{rechunk}({0: 50, 1: 1000}) \]

Automatic Chunking

Chunks also includes three special values:

1. \(-1\): no chunking along this dimension
2. \(\text{None}\): no change to the chunking along this dimension (useful for rechunk)
3. \"auto\": allow the chunking in this dimension to accommodate ideal chunk sizes

So, for example, one could rechunk a 3D array to have no chunking along the zeroth dimension, but still have sensible chunk sizes as follows:

\[ x = x.\text{rechunk}((0: -1, 1: \text{\textquotesingle} \text{\textquotesingle}auto\text{\textquotesingle}, 2: \text{\textquotesingle}auto\text{\textquotesingle})) \]

Or one can allow all dimensions to be auto-scaled to get to a good chunk size:

\[ x = x.\text{rechunk}('\text{\textquotesingle}auto\text{\textquotesingle}) \]

Automatic chunking expands or contracts all dimensions marked with \"auto\" to try to reach chunk sizes with a number of bytes equal to the config value \text{array.chunk-size}, which is set to 128MiB by default, but which you can change in your configuration.
Automatic rechunking tries to respect the median chunk shape of the auto-rescaled dimensions, but will modify this to accommodate the shape of the full array (can’t have larger chunks than the array itself) and to find chunk shapes that nicely divide the shape.

These values can also be used when creating arrays with operations like `dask.array.ones` or `dask.array.from_array`.

```python
>>> dask.array.ones((10000, 10000), chunks=(-1, 'auto'))
dask.array<wrapped, shape=(10000, 10000), dtype=float64, chunksize=(10000, 1250)>
```

### 4.7.4 Create Dask Arrays

You can load or store Dask arrays from a variety of common sources like HDF5, NetCDF, Zarr, or any format that supports NumPy-style slicing.

- `from_array(x[, chunks, name, lock, asarray, ...])` Create dask array from something that looks like an array.
- `from_delayed(value, shape, dtype[, name])` Create a dask array from a dask delayed value.
- `from_npy_stack(dirname[, mmap_mode])` Load dask array from stack of npy files.
- `from_zarr(url[, component, storage_options, ...])` Load array from the zarr storage format.
- `stack(seq[, axis])` Stack arrays along a new axis.
- `concatenate(seq[, axis, ...])` Concatenate arrays along an existing axis.

### NumPy Slicing

- `from_array(x[, chunks, name, lock, asarray, ...])` Create dask array from something that looks like an array.

Many storage formats have Python projects that expose storage using NumPy slicing syntax. These include HDF5, NetCDF, BColz, Zarr, GRIB, etc. For example, we can load a Dask array from an HDF5 file using h5py:

```python
>>> 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
```

Given an object like `d` above that has `dtype` and `shape` properties and that supports NumPy style slicing, we can construct a lazy Dask array:

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

This process is entirely lazy. Neither creating the h5py object nor wrapping it with `da.from_array` have loaded any data.
Random Data

For experimentation or benchmarking it is common to create arrays of random data. The dask.array.random module implements most of the functions in the numpy.random module. We list some common functions below but for a full list see the Array API:

- **random.binomial(n, p[, size])** Draw samples from a binomial distribution.
- **random.normal([loc, scale, size])** Draw random samples from a normal (Gaussian) distribution.
- **random.poisson([lam, size])** Draw samples from a Poisson distribution.
- **random.random([size])** Return random floats in the half-open interval [0.0, 1.0).

```python
>>> import dask.array as da
>>> x = da.random.random((10000, 10000), chunks=(1000, 1000))
```

Concatenation and Stacking

- **stack(seq[, axis])** Stack arrays along a new axis
- **concatenate(seq[, axis, ...])** Concatenate arrays along an existing axis

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

```python
dask_arrays = []
for fn in filenames:
    f = h5py.File(fn)
    d = f['/data']
    array = da.from_array(d, chunks=(1000, 1000))
    dask_arrays.append(array)

x = da.concatenate(dask_arrays, axis=0)  # concatenate arrays along first axis
```

For more information, see concatenation and stacking docs.

Using dask.delayed

- **from_delayed(value, shape, dtype[, name])** Create a dask array from a dask delayed value
- **stack(seq[, axis])** Stack arrays along a new axis
- **concatenate(seq[, axis, ...])** Concatenate arrays along an existing axis

Sometimes NumPy-style data resides in formats that do not support NumPy-style slicing. We can still construct Dask arrays around this data if we have a Python function that can generate pieces of the full array if we use dask.delayed. Dask delayed lets us delay a single function call that would create a NumPy array. We can then wrap this delayed object with da.from_delayed, providing a dtype and shape to produce a single-chunked Dask array. Furthermore, we can use stack or concatenate from before to construct a larger lazy array.

As an example, consider loading a stack of images using skimage.io.imread:

```python
import skimage.io
import dask.array as da
import dask
```

(continues on next page)
imread = dask.delayed(skimage.io.imread, pure=True)  # Lazy version of imread
filenames = sorted(glob.glob('*.jpg'))

lazy_images = [imread(path) for path in filenames]  # Lazily evaluate imread on each path
sample = lazy_images[0].compute()  # load the first image (assume rest are same shape/dtype)

arrays = [da.from_delayed(lazy_image, dtype=sample.dtype, shape=sample.shape) for lazy_image in lazy_images]
stack = da.stack(arrays, axis=0)  # Stack all small Dask arrays into one

See documentation on using dask.delayed with collections.

From Dask DataFrame

There are several ways to create a Dask array from a Dask DataFrame. Dask DataFrames have a to_dask_array method:

```python
>>> df = dask.dataframes.from_pandas(...)  
>>> df.to_dask_array()
```

dask.array<values, shape=(nan, 3), dtype=float64, chunksize=(nan, 3)>

This mirrors the to_numpy function in Pandas. The values attribute is also supported:

```python
>>> df.values
```

dask.array<values, shape=(nan, 3), dtype=float64, chunksize=(nan, 3)>

However, these arrays do not have known chunk sizes because dask.dataframe does not track the number of rows in each partition. This means that some operations like slicing will not operate correctly.

The chunk sizes can be computed:

```python
>>> df.to_dask_array(lengths=True)
```

dask.array<array, shape=(100, 3), dtype=float64, chunksize=(50, 3)>

Specifying lengths=True triggers immediate computation of the chunk sizes. This enables downstream computations that rely on having known chunk sizes (e.g., slicing).

The Dask DataFrame to_records method also returns a Dask Array, but does not compute the shape information:

```python
>>> df.to_records()
```

dask.array<to_records, shape=(nan,), dtype=(numpy.record, [('index', '<i8'), ('x', '<f8'), ('y', '<f8'), ('z', '<f8')]), chunksize=(nan,)>

If you have a function that converts a Pandas DataFrame into a NumPy array, then calling map_partitions with that function on a Dask DataFrame will produce a Dask array:

```python
>>> df.map_partitions(np.asarray)
```

dask.array<asarray, shape=(nan, 3), dtype=float64, chunksize=(nan, 3)>

350 Chapter 4. Index
Interactions with NumPy arrays

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

```python
>>> 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:

```python
>>> 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

See documentation on Array Chunks for more information.

4.7.5 Store Dask Arrays

- `store(sources, targets[, lock, regions,...])` Store dask arrays in array-like objects, overwrite data in target
- `to_hdf5(filename, *args, **kwargs)` Store arrays in HDF5 file
- `to_npy_stack(dirname, x[, axis])` Write dask array to a stack of .npy files
- `to_zarr(arr, url[, component,...])` Save array to the zarr storage format
- `compute(*args, **kwargs)` Compute several dask collections at once.

In Memory

- `compute(*args, **kwargs)` Compute several dask collections at once.

If you have a small amount of data, you can call `np.array` or `.compute()` on your Dask array to turn it into a normal NumPy array:

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

NumPy style slicing
**store**(sources, targets[, lock, regions, ...]) Store dask arrays in array-like objects, overwrite data in target

You can store Dask arrays in any object that supports NumPy-style slice assignment like h5py.Dataset:

```python
>>> import h5py
>>> f = h5py.File('myfile.hdf5')
>>> d = f.require_dataset('/data', shape=x.shape, dtype=x.dtype)
>>> da.store(x, d)
```

Also, you can store several arrays in one computation by passing lists of sources and destinations:

```python
>>> da.store([array1, array2], [output1, output2])  # doctest: +SKIP
```

### HDF5

to_hdf5(filename, *args, **kwargs) Store arrays in HDF5 file

HDF5 is sufficiently common that there is a special function to_hdf5 to store data into HDF5 files using h5py:

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

You can store several arrays in one computation with the function da.to_hdf5 by passing in a dictionary:

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

### Zarr

The Zarr format is a chunk-wise binary array storage file format with a good selection of encoding and compression options. Due to each chunk being stored in a separate file, it is ideal for parallel access in both reading and writing (for the latter, if the Dask array chunks are aligned with the target). Furthermore, storage in remote data services such as S3 and GCS is supported.

For example, to save data to a local zarr dataset you would do:

```python
>>> arr.to_zarr('output.zarr')
```

or to save to a particular bucket on S3:

```python
>>> arr.to_zarr('s3://mybucket/output.zarr', storage_option={'key': 'mykey', 'secret': 'mysecret'})
```

or your own custom zarr Array:

```python
>>> z = zarr.create((10,), dtype=float, store=zarr.ZipStore('output.zarr'))
>>> arr.to_zarr(z)
```

To retrieve those data, you would do da.from_zarr with exactly the same arguments. The chunking of the resultant Dask array is defined by how the files were saved, unless otherwise specified.
TileDB

TileDB is a binary array format and storage manager with tunable chunking, layout, and compression options. The TileDB storage manager library includes support for scalable storage backends such as S3 API compatible object stores and HDFS, with automatic scaling, and supports multi-threaded and multi-process reads (consistent) and writes (eventually-consistent).

To save data to a local TileDB array:

```python
>>> arr.to_tiledb('output.tdb')
```

or to save to a bucket on S3:

```python
>>> arr.to_tiledb('s3://mybucket/output.tdb',
    storage_options={'vfs.s3.aws_access_key_id': 'mykey',
                     'vfs.s3.aws_secret_access_key': 'mysecret'})
```

Files may be retrieved by running `da.from_tiledb` with the same URI, and any necessary arguments.

Intermediate storage

<table>
<thead>
<tr>
<th><code>store</code></th>
<th>Store dask arrays in array-like objects, overwrite data in target</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>store(source[, targets], lock, regions, ...)</code></td>
<td></td>
</tr>
</tbody>
</table>

In some cases, one may wish to store an intermediate result in long term storage. This differs from `persist`, which is mainly used to manage intermediate results within Dask that don’t necessarily have longevity. Also it differs from storing final results as these mark the end of the Dask graph. Thus intermediate results are easier to reuse without reloading data. Intermediate storage is mainly useful in cases where the data is needed outside of Dask (e.g. on disk, in a database, in the cloud, etc.). It can be useful as a checkpoint for long running or error-prone computations.

The intermediate storage use case differs from the typical storage use case as a Dask Array is returned to the user that represents the result of that storage operation. This is typically done by setting the `store` function’s `return_stored` flag to `True`.

```python
x.store() # stores data, returns nothing
x = x.store(return_stored=True) # stores data, returns new dask array backed by that data
```

The user can then decide whether the storage operation happens immediately (by setting the `compute` flag to `True`) or later (by setting the `compute` flag to `False`). In all other ways, this behaves the same as a normal call to `store`. Some examples are shown below.

```python
>>> import dask.array as da
>>> import zarr as zr
>>> c = (2, 2)
>>> d = da.ones((10, 11), chunks=c)
>>> zl = zr.open_array('lazy.zarr', shape=d.shape, dtype=d.dtype, chunks=c)
>>> z2 = zr.open_array('eager.zarr', shape=d.shape, dtype=d.dtype, chunks=c)
>>> d1 = d.store(zl, compute=False, return_stored=True)
>>> d2 = d.store(z2, compute=True, return_stored=True)
```

This can be combined with any other storage strategies either noted above, in the docs or for any specialized storage types.
4.7.6 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:

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

```python
>>> with dask.config.set(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:

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

```python
>>> 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:

```python
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")
```

```python
with dask.config.set(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:

```python
with dask.config.set(array_plugins=[warn_on_large_chunks, lambda x: x.compute()]):
...
```
4.7.7 Overlapping Computations

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 embarrassingly parallel manner.

The main API for these computations is the `map_overlap` method defined below:

```python
map_overlap(x, func, depth[, boundary, trim]) Map a function over blocks of the array with some overlap
```

dask.array.map_overlap(x, 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 elements that each block should share with its neighbors. If a tuple or dict then 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 `depth` elements from each block after calling the map function. Set this to False if your mapping function already does this for you
- **kwargs**: Other keyword arguments valid in `map_blocks`

**Examples**

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

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

4.7. Array
>>> 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()  # doctest: +NORMALIZE_
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27]])

Explanation

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 overlap 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.overlap.overlap(d, depth={0: 2, 1: 1},
... boundary={0: 100, 1: 'reflect'})

(continues on next page)
```
>>> 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,   5,   6,   7,   7,   7],
      [  8,   8,   9,  10,  11,  12,  13,  14,  15,  15,  15],
      [ 16,  16,  17,  18,  19,  20,  21,  22,  23,  23,  23],
      [ 24,  24,  25,  26,  27,  28,  29,  30,  31,  31,  31],
      [ 32,  32,  33,  34,  35,  36,  37,  38,  39,  39,  39],
      [ 40,  40,  41,  42,  43,  44,  45,  46,  47,  47,  47],
      [ 16,  16,  17,  18,  19,  20,  21,  22,  23,  23,  23],
      [ 24,  24,  25,  26,  27,  28,  29,  30,  31,  31,  31],
      [ 32,  32,  33,  34,  35,  36,  37,  38,  39,  39,  39],
      [ 40,  40,  41,  42,  43,  44,  45,  46,  47,  47,  47],
      [ 48,  48,  49,  50,  51,  52,  53,  54,  55,  55,  55],
      [ 56,  56,  57,  58,  59,  60,  61,  62,  63,  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**

With respect to overlapping, 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

An example boundary kind argument might look like the following:

```
{0: 'periodic',
 1: 'reflect',
 2: np.nan}
```

Alternatively, you can use `dask.array.pad()` for other types of paddings.

**Map a function across blocks**

Overlapping 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, any arbitrary function could have been used instead. 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 size is regular, then this argument can take a block shape of, for example, `(1000, 1000)`. In case of irregular block sizes, it must be a tuple with the full chunks shape like `((1000, 700, 1000), (200, 300))`.  

**4.7. Array**
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`:

```python
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-left 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 `overlap`:

```python
>>> x.chunks
((10, 10, 10, 10), (10, 10, 10, 10))

>>> y = da.overlap.trim_internal(x, {0: 2, 1: 1})
>>> y.chunks
((6, 6, 6, 6), (8, 8, 8, 8))
```

**Full Workflow**

And so, a pretty typical overlapping workflow includes `overlap`, `map_blocks` and `trim_internal`:

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

>>> g = da.overlap.overlap(x, depth={0: 2, 1: 2},
...                           boundary={0: 'periodic', 1: 'periodic'})

>>> g2 = g.map_blocks(myfunc)

>>> result = da.overlap.trim_internal(g2, {0: 2, 1: 2})
```

**4.7.8 Internal Design**

**Overview**

```plaintext
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>(x', 0, 0)</td>
<td></td>
<td>(x', 0, 1)</td>
</tr>
<tr>
<td></td>
<td>(x', 1, 0)</td>
<td></td>
<td>(x', 1, 1)</td>
</tr>
<tr>
<td></td>
<td>(x', 2, 0)</td>
<td></td>
<td>(x', 2, 1)</td>
</tr>
<tr>
<td></td>
<td>(x', 3, 0)</td>
<td></td>
<td>(x', 3, 1)</td>
</tr>
</tbody>
</table>
```
Dask arrays define a large array with a grid of blocks of smaller arrays. These arrays may be actual arrays 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

```python
>>> 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), with 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. Moreover, it likely also holds other key-value pairs required to eventually compute the desired values:

```python
[
    ('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 .__dask_keys__() method. Additionally, 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 composed of a tuple of tuples such that the length of the outer tuple is equal to the number of dimensions of the array, 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:

\[
\text{chunks} = ((5, 5, 5, 5), (8, 8, 8))
\]

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 \texttt{chunks} reflects properties of our array:

1. \texttt{len(x.chunks) == x.ndim} the length of chunks is the number of dimensions
2. \texttt{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 \texttt{chunks} == ((a, b), (d, e, f)) and \texttt{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**

In order to create an \texttt{da.Array} object we need a dictionary with these special keys:

\[
dsk = {('x', 0, 0): ...}
\]

a name specifying which keys this array refers to:

\[
\text{name} = 'x'
\]

and a chunks tuple:

\[
\text{chunks} = ((5, 5, 5, 5), (8, 8, 8))
\]

Then, using these elements, one can construct an array:

\[
x = da.Array(dsk, name, chunks)
\]

In short, \texttt{dask.array} operations update Dask graphs, update dtypes, and track chunk shapes.

**Example - eye function**

As an example, lets build the \texttt{np.eye} function for \texttt{dask.array} to make the identity matrix:

\[
def\text{eye}(n, blocksize):
    \text{chunks} = ((\text{blocksize},) \times (n // \text{blocksize}),
    (\text{blocksize},) \times (n // \text{blocksize}))

    \text{name} = 'eye' + \text{next(tokens)} \quad \# \text{unique identifier}

    \text{dsk} = {\text{name}, i, j: (\text{np.eye}, \text{blocksize})}
\]

(continues on next page)
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)

4.7.9 Sparse Arrays

By swapping out in-memory NumPy arrays with in-memory sparse arrays, we can reuse the blocked algorithms of
Dask’s 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:

```python
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:

```python
import sparse
s = x.map_blocks(sparse.COO)
```

Now, our array is not composed of many NumPy arrays, but rather of many sparse arrays. Semantically, this does not
change anything. Operations that work will continue to work identically (assuming that the behavior of numpy and
sparse are identical), but performance characteristics and storage costs may change significantly:

```python
>>> 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,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485,
       4938.94446802, 4849.51326473, 4858.83977856, 4847.81468485],
       dtype=object)
```

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’s 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.).

### 4.7.10 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.

```python
>>> 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.

```python
>>> a = da.random.uniform(size=(50,), chunks=(25,))
>>> b = a + da.random.uniform(low=-0.15, high=0.15, size=(50,), chunks=(25,))
>>> result = stats.ttest_rel(a, b)
>>> result.compute()
Ttest_relResult(statistic=-1.5102104380013242, pvalue=0.13741197274874514)
```

### 4.7.11 LinearOperator

Dask Array implements the SciPy `LinearOperator` interface and it can be used with any SciPy algorithm depending on that interface.
Example

```python
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.

### 4.7.12 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]]`
- Slicing one `Array` with an `Array` of bools: `x[x > 0]`
- Slicing one `Array` with a zero or one-dimensional `Array` of ints: `a[b.argtopk(5)]`

However, it does not currently support the following:

- Slicing with lists in multiple axes: `x[[1, 2, 3], [3, 2, 1]]`
  This is straightforward to add though. If you have a use case then raise an issue. Also, users interested in this should take a look at `vindex`.
- Slicing one `Array` with a multi-dimensional `Array` of ints

#### Efficiency

The normal Dask schedulers are smart enough to compute only those blocks that are necessary to achieve the desired slicing. Hence, large operations may be cheap if only a small output is desired.

In the example below, we create a Dask array with a trillion elements with million element sized blocks. We then operate on the entire array and finally slice out only a portion of the output:

```python
>>> # 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 slightly wasteful on those blocks where we need only partial results. Moreover, 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.
4.7.13 Stack, Concatenate, and Block

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`, `da.concatenate`, and `da.block`.

**Stack**

We stack many existing Dask arrays into a new array, creating a new dimension as we go.

```python
>>> import dask.array as da

>>> arr0 = da.from_array(np.zeros((3, 4)), chunks=(1, 2))
>>> arr1 = da.from_array(np.ones((3, 4)), chunks=(1, 2))

>>> data = [arr0, arr1]

>>> x = da.stack(data, axis=0)
>>> x.shape
(2, 3, 4)

>>> da.stack(data, axis=1).shape
(3, 2, 4)

>>> da.stack(data, axis=-1).shape
(3, 4, 2)
```

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

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

>>> arr0 = da.from_array(np.zeros((3, 4)), chunks=(1, 2))
>>> arr1 = da.from_array(np.ones((3, 4)), chunks=(1, 2))

>>> data = [arr0, arr1]

>>> x = da.concatenate(data, axis=0)
>>> x.shape
(6, 4)

>>> da.concatenate(data, axis=1).shape
(3, 8)
```

**Block**

We can handle a larger variety of cases with `da.block` as it allows concatenation to be applied over multiple dimensions at once. This is useful if your chunks tile a space, for example if small squares tile a larger 2-D plane.
4.7.14 Generalized Ufuncs

EXPERIMENTAL FEATURE added to Version 0.18.0 and above - see disclaimer.

NumPy provides the concept of generalized ufuncs. Generalized ufuncs are functions that distinguish the various dimensions of passed arrays in the two classes loop dimensions and core dimensions. To accomplish this, a signature is specified for NumPy generalized ufuncs.

Dask integrates interoperability with NumPy’s generalized ufuncs by adhering to respective ufunc protocol, and provides a wrapper to make a Python function a generalized ufunc.

Usage

NumPy generalized ufunc

Note: NumPy generalized ufuncs are currently (v1.14.3 and below) stored in inside np.linalg._umath_linalg and might change in the future.

Wrap own Python function

gufunc can be used to make a Python function behave like a generalized ufunc:

(continues on next page)
Instead of `gufunc`, also the `as_gufunc` decorator can be used for convenience:

```python
def gufoo(x):
    return np.mean(x, axis=-1)
```

```python
x = da.random.normal(size=(10, 5), chunks=(2, 5))

@dask.gufunc(signature=’(i)->()’, output_dtypes=’float’, vectorize=True)
def gufoo(x):
    return np.mean(x, axis=-1)
```

```
y = gufoo(x)
```

**Disclaimer**

This experimental generalized ufunc integration is not complete:

- `gufunc` does not create a true generalized ufunc to be used with other input arrays besides Dask. I.e., at the moment, `gufunc` casts all input arguments to `dask.array.Array`
- Inferring `output_dtypes` automatically is not implemented yet

**API**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply_gufunc(func, signature, *args, **kwargs)</code></td>
<td>Apply a generalized ufunc or similar python function to arrays.</td>
</tr>
<tr>
<td><code>as_gufunc([signature])</code></td>
<td>Decorator for <code>dask.array.gufunc</code>.</td>
</tr>
<tr>
<td><code>gufunc(pyfunc, **kwargs)</code></td>
<td>Binds <code>pyfunc</code> into <code>dask.array.apply_gufunc</code> when called.</td>
</tr>
</tbody>
</table>

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.

**4.7.15 Design**

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

**4.7.16 Common Uses**

Dask Array is used in fields like atmospheric and oceanographic science, large scale imaging, genomics, numerical algorithms for optimization or statistics, and more.

**4.7.17 Scope**

Dask arrays support most of the NumPy interface like the following:
• 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]]
• Array protocols like __array__ and __array_ufunc__
• Some linear algebra: svd, qr, solve, solve_triangular, lstsq
• ...

However, Dask Array does not implement the entire NumPy interface. Users expecting this will be disappointed. Notably, Dask Array lacks the following features:

• Much of np.linalg has not been implemented. This has been done by a number of excellent BLAS/LAPACK implementations, and is the focus of numerous ongoing academic research projects
• Arrays with unknown shapes do not support all operations
• 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
• 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
• Dask development is driven by immediate need, hence many lesser used functions have not been implemented. Community contributions are encouraged

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

4.7.18 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.

4.8 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.

4.8.1 Overview

Dask Bag implements operations like map, filter, fold, and groupby on collections of Python objects. It does this in parallel with a small memory footprint 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)
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’s `groupby` is slow. You should try to use Bag’s `foldby` if possible. Using `foldby` requires more thought tough

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.

## 4.8.2 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:

```python
>>> 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:

```python
>>> 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 a Dask bag. Instead, use Dask Bag to load your data. This parallelizes the loading step and reduces inter-worker communication:

```python
>>> b = db.from_sequence(['1.dat', '2.dat', ...]).map(load_from_filename)
```

### db.read_text

Dask Bag can load data directly from text files. You can pass either a single file name, a list of file names, or a globstring. The resulting bag will have one item per line and one file per partition:

```python
>>> 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 the file name extension, or by using the `compression='gzip'` keyword:

```python
>>> 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:

```python
>>> 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`:

```python
>>> b = db.read_text('myfile.*.csv').str.strip().str.split(',')
```
db.read_avro

Dask Bag can read binary files in the Avro format if fastavro is installed. A bag can be made from one or more files, with optional chunking within files. The resulting bag will have one item per Avro record, which will be a dictionary of the form given by the Avro schema. There will be at least one partition per input file:

```python
>>> b = db.read_avro('datafile.avro')
```

```python
>>> b = db.read_avro('data.*.avro')
```

By default, Dask will split data files into chunks of approximately blocksize bytes in size. The actual blocks you would get depend on the internal blocking of the file.

For files that are compressed after creation (this is not the same as the internal “codec” used by Avro), no chunking should be used, and there will be exactly one partition per file:

```python
>>> b = bd.read_avro('compressed.*.avro.gz', blocksize=None, compression='gzip')
```

db.from_delayed

You can construct a Dask bag from dask.delayed values using the db.from_delayed function. For more information, see documentation on using dask.delayed with collections.

4.8.3 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:

```python
>>> result = b.compute()
```

or

```python
>>> result = list(b)
```

To Text Files

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, storage_options=None, last_endline=False, **kwargs)

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

```python
>>> b.to_textfiles('/path/to/data/*.json.gz')  # doctest: +SKIP
```

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.

```python
>>> 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)  # doctest: +SKIP
/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.

```python
>>> paths = ['/path/to/data/alice.json.gz', '/path/to/data/bob.json.gz', ...]  # doctest: +SKIP

>>> b.to_textfiles(paths)  # doctest: +SKIP
```

**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`:

```python
>>> b_dict.map(json.dumps).to_textfiles('/path/to/data/*.json')  # doctest: +SKIP
```

**Last endline**: By default the last line does not end with a newline character. Pass `last_endline=True` to invert the default.

### To Avro

Dask bags can be written directly to Avro binary format using `fastavro`. One file will be written per bag partition. This requires the user to provide a fully-specified schema dictionary (see the docstring of the `.to_avro()` method).

```python
dask.bag.avro.to_avro(b, filename, schema=None, name_function=None, storage_options=None, codec='null', sync_interval=16000, metadata=None, compute=True, **kwargs)
```

Write bag to set of avro files

The schema is a complex dictionary describing the data, see [https://avro.apache.org/docs/1.8.2/gettingstartedpython.html#Defining+a+schema](https://avro.apache.org/docs/1.8.2/gettingstartedpython.html#Defining+a+schema) and [https://fastavro.readthedocs.io/en/latest/writer.html](https://fastavro.readthedocs.io/en/latest/writer.html). It's structure is as follows:

```python
{ 'name': 'Test',
  'namespace': 'Test',
  'doc': 'Descriptive text',
  'type': 'record',
}
```

(continues on next page)
where the “name” field is required, but “namespace” and “doc” are optional descriptors; “type” must always be “record”. The list of fields should have an entry for every key of the input records, and the types are like the primitive, complex or logical types of the Avro spec (https://avro.apache.org/docs/1.8.2/spec.html).

Results in one avro file per input partition.

**Parameters**

- **b**: dask.bag.Bag
- **filename**: list of str or str
  - Filenames to write to. If a list, number must match the number of partitions. If a string, must include a glob character “*”, which will be expanded using name_function
- **schema**: dict
  - Avro schema dictionary, see above
- **name_function**: None or callable
  - Expands integers into strings, see dask.bytes.utils.build_name_function
- **storage_options**: None or dict
  - Extra key/value options to pass to the backend file-system
- **codec**: ‘null’, ‘deflate’, or ‘snappy’
  - Compression algorithm
- **sync_interval**: int
  - Number of records to include in each block within a file
- **metadata**: None or dict
  - Included in the file header
- **compute**: bool
  - If True, files are written immediately, and function blocks. If False, returns delayed objects, which can be computed by the user where convenient.
- **kwargs**: passed to compute(), if compute=True

**Examples**

```python
>>> import dask.bag as db
>>> b = db.from_sequence([('Alice', 100), ...
...                       ('Bob', 200)])
>>> schema = {'name': 'People', 'doc': "Set of people's scores",
...           'type': 'record', ...
...           'fields': [(('name', 'name', 'type': 'string'),
...                       ('name', 'value', 'type': 'int')]
>>> b.to_avro('my-data.*.avro', schema)  # doctest: +SKIP
['my-data.0.avro', 'my-data.1.avro']
```

**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

```python
>>> 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(optimize_graph=True)

Convert into a list of dask.delayed objects, one per partition.

Parameters

- **optimize_graph** [bool, optional] If True [default], the graph is optimized before converting into dask.delayed objects.

See also:

- `dask.bag.from_delayed`

4.8.4 API

Top level user functions:

- **Bag** (dsk, name, npartitions) Parallel collection of Python objects
- **Bag.all** ([split_every]) Are all elements truthy?

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Bag.any([split_every])</code></td>
<td>Are any of the elements truthy?</td>
</tr>
<tr>
<td><code>Bag.compute(**kwargs)</code></td>
<td>Compute this dask collection</td>
</tr>
<tr>
<td><code>Bag.count([split_every])</code></td>
<td>Count the number of elements.</td>
</tr>
<tr>
<td><code>Bag.distinct([key])</code></td>
<td>Distinct elements of collection</td>
</tr>
<tr>
<td><code>Bag.filter(predicate)</code></td>
<td>Filter elements in collection by a predicate function.</td>
</tr>
<tr>
<td><code>Bag.flatten()</code></td>
<td>Concatenate nested lists into one long list.</td>
</tr>
<tr>
<td><code>Bag.fold(binop[, combine, initial, ...])</code></td>
<td>Parallelizable reduction</td>
</tr>
<tr>
<td><code>Bag.foldby(key, binop[, initial, combine, ...])</code></td>
<td>Combined reduction and groupby.</td>
</tr>
<tr>
<td><code>Bag.frequencies([split_every, sort])</code></td>
<td>Count number of occurrences of each distinct element.</td>
</tr>
<tr>
<td><code>Bag.groupby(grouper[, method, npartitions, ...])</code></td>
<td>Group collection by key function</td>
</tr>
<tr>
<td><code>Bag.join(other, on_self[, on_other])</code></td>
<td>Joins collection with another collection.</td>
</tr>
<tr>
<td><code>Bag.map(func, *args, **kwargs)</code></td>
<td>Apply a function elementwise across one or more bags.</td>
</tr>
<tr>
<td><code>Bag.map_partitions(func, *args, **kwargs)</code></td>
<td>Apply a function to every partition across one or more bags.</td>
</tr>
<tr>
<td><code>Bag.max([split_every])</code></td>
<td>Maximum element</td>
</tr>
<tr>
<td><code>Bag.mean()</code></td>
<td>Arithmetic mean</td>
</tr>
<tr>
<td><code>Bag.min([split_every])</code></td>
<td>Minimum element</td>
</tr>
<tr>
<td><code>Bagpluck(key[, default])</code></td>
<td>Select item from all tuples/dicts in collection.</td>
</tr>
<tr>
<td><code>Bag.product(other)</code></td>
<td>Cartesian product between two bags.</td>
</tr>
<tr>
<td><code>Bag.reduction(perpartition, aggregate[,...])</code></td>
<td>Reduce collection with reduction operators.</td>
</tr>
<tr>
<td><code>Bag.random_sample(prob[, random_state])</code></td>
<td>Return elements from bag with probability of prob.</td>
</tr>
<tr>
<td><code>Bag.remove(predicate)</code></td>
<td>Remove elements in collection that match predicate.</td>
</tr>
<tr>
<td><code>Bag.repartition(npartitions)</code></td>
<td>Coalesce bag into fewer partitions.</td>
</tr>
<tr>
<td><code>Bag.starmap(func, *npartitions)</code></td>
<td>Apply a function using argument tuples from the given bag.</td>
</tr>
<tr>
<td><code>Bag.std([ddof])</code></td>
<td>Standard deviation</td>
</tr>
<tr>
<td><code>Bag.sum([split_every])</code></td>
<td>Sum all elements</td>
</tr>
<tr>
<td><code>Bag.take(k[, npartitions, compute, warn])</code></td>
<td>Take the first k elements.</td>
</tr>
<tr>
<td><code>Bag.to_dataframe([meta, columns])</code></td>
<td>Create Dask Dataframe from a Dask Bag.</td>
</tr>
<tr>
<td><code>Bag.to_delayed([optimize_graph])</code></td>
<td>Convert into a list of dask.delayed objects, one per partition.</td>
</tr>
<tr>
<td><code>Bag.to_textfiles(path[, name_function, ...])</code></td>
<td>Write dask Bag to disk, one filename per partition, one line per element.</td>
</tr>
<tr>
<td><code>Bag.to_avro(filename, schema[,...])</code></td>
<td>Write bag to set of avro files</td>
</tr>
<tr>
<td><code>Bag.top(k[, key, split_every])</code></td>
<td>K largest elements in collection</td>
</tr>
<tr>
<td><code>Bag.var([ddof])</code></td>
<td>Variance</td>
</tr>
<tr>
<td><code>Bag.visualize([filename, format, optimize_graph])</code></td>
<td>Render the computation of this object’s task graph using graphviz.</td>
</tr>
</tbody>
</table>

### Create Bags

- `from_sequence(seq[, partition_size, npartitions])` Create a dask Bag from Python sequence.
- `from_delayed(values)` Create bag from many dask Delayed objects.
- `read_text(urlpath[, blocksize, compression,...])` Read lines from text files
- `from_url(urls)` Create a dask Bag from a url.
- `read_avro(urlpath[, blocksize,...])` Read set of avro files
- `range(n, npartitions)` Numbers from zero to n
Top-level functions

- `concat(bags)` Concatenate many bags together, unioning all elements.
- `map(func, *args, **kwargs)` Apply a function elementwise across one or more bags.
- `map_partitions(func, *args, **kwargs)` Apply a function to every partition across one or more bags.
- `zip(*bags)` Partition-wise bag zip

Turn Bags into other things

- `Bag.to_textfiles(path[, name_function, ...])` Write dask Bag to disk, one filename per partition, one line per element.
- `Bag.to_dataframe([meta, columns])` Create Dask Dataframe from a Dask Bag.
- `Bag.to_delayed([optimize_graph])` Convert into a list of dask.delayed objects, one per partition.
- `Bag.to_avro(filename, schema[, ...])` Write bag to set of avro files

Bag methods

```python
class dask.bag.Bag(dsk, name, npartitions)
```
Parallel collection of Python objects

Examples

Create Bag from sequence

```python
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.filter(lambda x: x % 2 == 0).map(lambda x: x * 10))
# doctest: +SKIP
[0, 20, 40]
```

Create Bag from filename or globstring of filenames

```python
>>> b = db.read_text('/path/to/mydata.*.json.gz').map(json.loads)
# doctest: +SKIP
```

Create manually (expert use)

```python
>>> dsk = {('x', 0): (range, 5),
...        ('x', 1): (range, 5),
...        ('x', 2): (range, 5)}
>>> b = Bag(dsk, 'x', npartitions=3)
```

```python
>>> sorted(b.map(lambda x: x * 10))
# doctest: +SKIP
[0, 0, 0, 10, 10, 10, 20, 20, 20, 30, 30, 30, 40, 40, 40]
```

```python
>>> int(b.fold(lambda x, y: x + y))
# doctest: +SKIP
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**

```python
>>> from operator import add
>>> b = from_sequence([1, 2, 3, 4, 5], npartitions=2)
>>> b.accumulate(add).compute()  # doctest: +SKIP
[1, 3, 6, 10, 15]
```

Accumulate also takes an optional argument that will be used as the first value.

```python
>>> b.accumulate(add, initial=-1)  # doctest: +SKIP
[-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** *(key=None)*

Distinct elements of collection

Unordered without repeats.

**Parameters**

* key: {callable,str} Defines uniqueness of items in bag by calling key on each item. If a string is passed key is considered to be lambda x: x[key].

**Examples**

```python
>>> b = from_sequence(['Alice', 'Bob', 'Alice'])
>>> sorted(b.distinct())
['Alice', 'Bob']
>>> b = from_sequence([{'name': 'Alice'}, {'name': 'Bob'}, {'name': 'Alice'}])
>>> b.distinct(key=lambda x: x['name']).compute()
[{'name': 'Alice'}, {'name': 'Bob'}]
>>> b.distinct(key='name').compute()
[{'name': 'Alice'}, {'name': 'Bob'}]
```

**filter** *(predicate)*

Filter elements in collection by a predicate function.

```python
>>> def iseven(x):
...     return x % 2 == 0

>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.filter(iseven))  # doctest: +SKIP
[0, 2, 4]
```
**flatten()**

Concatenate nested lists into one long list.

```python
>>> b = from_sequence([[1], [2, 3]])
>>> list(b)
[[1], [2, 3]]
```

```python
>>> list(b.flatten())
[1, 2, 3]
```

**fold**(  
`binop`, `combine=None`, `initial='__no__default__'`, `split_every=None`, `out_type=<class 'dask.bag.core.Item'>`  
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:

```python
>>> intermediates = [reduce(binop, part) for part in partitions]  # doctest: +SKIP
>>> final = reduce(combine, intermediates)  # doctest: +SKIP
```

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:

```python
>>> def add(x, y):
...     return x + y

>>> b = from_sequence(range(5))
>>> b.fold(add).compute()  # doctest: +SKIP
10
```

In full form we provide both binary operators as well as their default arguments

```python
>>> b.fold(binop=add, combine=add, initial=0).compute()  # doctest: +SKIP
10
```

More complex binary operators are also doable

```python
>>> 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()  # doctest: +SKIP
{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
is equivalent to the following:

```python
>>> def reduction(group):
...    return reduce(binop, group, init)
# doctest: +SKIP
```

But uses minimal communication and so is much faster.

```python
>>> b = from_sequence(range(10))
>>> iseven = lambda x: x % 2 == 0
>>> add = lambda x, y: x + y
>>> dict(b.foldby(iseven, add))
# doctest: +SKIP
{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.

```python
>>> 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:

```python
>>> b.foldby(
lambda x: x['name'], ...
# doctest: +SKIP
>>> b.foldby('name', ...)
# doctest: +SKIP
```

### 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:

```python
>>> def binop(total, x):
...    return total + x['amount']
```

Combine takes two totals and combines them:

```python
>>> 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.

```python
>>> b.foldby('name', binop, 0, combine, 0)
# doctest: +SKIP
```

### See also:
- `toolz.reduceby`
- `pyspark.combineByKey`
frequencies (split_every=None, sort=False)
Count number of occurrences of each distinct element.

```python
>>> b = from_sequence(['Alice', 'Bob', 'Alice'])
>>> dict(b.frequencies())  # doctest: +SKIP
{'Alice': 2, 'Bob', 1}
```

groupby (grouper, method=None, npartitions=None, blocksize=1048576, max_branch=None, shuffle=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

shuffle: 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

```python
>>> b = from_sequence(range(10))
>>> iseven = lambda x: x % 2 == 0
>>> dict(b.groupby(iseven))  # doctest: +SKIP
{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 one of the following:

1. An iterable. We recommend tuples over lists for internal performance reasons.
2. A delayed object, pointing to a tuple. This is recommended if the other collection is sizable and you’re using the distributed scheduler. Dask is able to pass around data wrapped in delayed objects with greater sophistication.
3. A Bag with a single partition

You might also consider Dask Dataframe, whose join operations are much more heavily optimized.

Parameters

other: Iterable, Delayed, Bag Other collection on which to join

on_self: callable Function to call on elements in this collection to determine a match

on_other: callable (defaults to on_self) Function to call on elements in the other collection to determine a match
Examples

```python
>>> people = from_sequence(['Alice', 'Bob', 'Charlie'])
>>> fruit = ['Apple', 'Apricot', 'Banana']
>>> list(people.join(fruit, lambda x: x[0]))
# doctest: +SKIP
['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/kwargs 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

```python
>>> 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:

```python
>>> b.map(lambda x: x + 1).compute()
[1, 2, 3, 4, 5]
```

Apply a function with arguments from multiple bags:

```python
>>> 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:

```python
>>> b.map(add, 1).compute()
[1, 2, 3, 4, 5]
```

Keyword arguments are also supported, and have the same semantics as regular arguments:

```python
>>> 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:

```python
>>> 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] The function to be called on every partition. This function should expect an `Iterator` or `Iterable` for every partition and should return an `Iterator` or `Iterable` in return.

- **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**

```python
>>> 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:

```python
>>> 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:

```python
>>> 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.
>>> list(b.pluck('name')) # doctest: +SKIP
['Alice', 'Bob']

>>> list(b.pluck('credits').pluck(0)) # doctest: +SKIP
[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**

```python
>>> import dask.bag as db

>>> b = db.from_sequence(range(5))

>>> list(b.random_sample(0.5, 42))
[1, 3]
```

**reduction** *(perpartition, aggregate, split_every=None, out_type=<class 'dask.bag.core.Item'>, name=None)*

Reduce collection with reduction operators.

**Parameters**

- **perpartition**: function reduction to apply to each partition
- **aggregate**: function reduction to apply to the results of all partitions
- **split_every**: int (optional) Group partitions into groups of this size while performing reduction. Defaults to 8
- **out_type**: {Bag, Item} The out type of the result, Item if a single element, Bag if a list of elements. Defaults to Item.

**Examples**

```python
>>> b = from_sequence(range(10))

>>> b.reduction(sum, sum).compute()
45
```

**remove** *(predicate)*

Remove elements in collection that match predicate.

```python
>>> def iseven(x):
...     return x % 2 == 0
```
>>> import dask.bag as db
>>> b = db.from_sequence(range(5))
>>> list(b.remove(iseven))  # doctest: +SKIP
[1, 3]

**repartition**(npartitions)

Coalesce bag into fewer partitions.

**Examples**

>>> b.repartition(5)  # set to have 5 partitions  # doctest: +SKIP

**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:

```python
>>> 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**

```python
>>> 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:

```python
>>> from operator import add
>>> b.starmap(add).compute()
[3, 7, 11, 15, 19]
```

Apply a function to each argument tuple, with additional keyword arguments:

```python
>>> 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`:

```python
>>> max_second = b.pluck(1).max()
>>> max_second.compute()
10
>>> b.starmap(myadd, z=max_second).compute()
[13, 17, 21, 25, 29]
```

4.8. Bag
std(ddof=0)
    Standard deviation

str
    String processing functions

Examples

```python
>>> 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.

```python
>>> b = from_sequence(range(10))
>>> b.take(3) # doctest: +SKIP
(0, 1, 2)
```

to_avro(filename, schema, name_function=None, storage_options=None, codec='null',
    sync_interval=16000, metadata=None, compute=True, **kwargs)
    Write bag to set of avro files

The schema is a complex dictionary describing the data, see https://avro.apache.org/docs/1.8.2/gettingstartedpython.html#Defining+a+schema and https://fastavro.readthedocs.io/en/latest/writer.html. It's structure is as follows:

```json
{ 'name': 'Test',
  'namespace': 'Test',
  'doc': 'Descriptive text',
  'type': 'record',
  'fields': [ { 'name': 'a', 'type': 'int' },
      ]
}
```
where the “name” field is required, but “namespace” and “doc” are optional descriptors; “type” must always be “record”. The list of fields should have an entry for every key of the input records, and the types are like the primitive, complex or logical types of the Avro spec (https://avro.apache.org/docs/1.8.2/spec.html).

Results in one avro file per input partition.

Parameters

- **b**: dask.bag.Bag
- **filename**: list of str or str  Filenames to write to. If a list, number must match the number of partitions. If a string, must includ a glob character “*”, which will be expanded using name_function
- **schema**: dict  Avro schema dictionary, see above
- **name_function**: None or callable  Expands integers into strings, see dask.bytes.utils.build_name function
- **storage_options**: None or dict  Extra key/value options to pass to the backend file-system
- **codec**: ‘null’, ‘deflate’, or ‘snappy’  Compression algorithm
- **sync_interval**: int  Number of records to include in each block within a file
- **metadata**: None or dict  Included in the file header
- **compute**: bool  If True, files are written immediately, and function blocks. If False, returns delayed objects, which can be computed by the user where convenient.
- **kwargs**: passed to compute(), if compute=True

Examples

```
>>> import dask.bag as db

>>> b = db.from_sequence([{'name': 'Alice', 'value': 100},
...                       {'name': 'Bob', 'value': 200}])

>>> schema = {'name': 'People', 'doc': "Set of people's scores",
...           'type': 'record',
...           'fields': [
...                       {'name': 'name', 'type': 'string'},
...                       {'name': 'value', 'type': 'int'}
...           ]}

>>> b.to_avro('my-data.*.avro', schema)  # doctest: +SKIP
['my-data.0.avro', 'my-data.1.avro']
```

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**

```python
>>> 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
```

**toDelayed** *(optimize_graph=True)*

Convert into a list of dask.delayed objects, one per partition.

**Parameters**

- **optimize_graph** [bool, optional] If True [default], the graph is optimized before converting into dask.delayed objects.

**See also:**

dask.bag.from_delayed
dask.bag.to_textfiles *(path, name_function=None, compression='infer', encoding='utf-8', compute=True, storage_options=None, last_endline=False, **kwargs)*

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

```python
>>> b.to_textfiles('/path/to/data/*.json.gz')  # doctest: +SKIP

/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.

```python
>>> from datetime import date, timedelta

>>> def name(i):
...     return str(date(2015, 1, 1) + i * timedelta(days=1))
```

```python
386 Chapter 4. Index
```
You can also provide an explicit list of paths.

```python
>>> paths = ["/path/to/data/alice.json.gz", "/path/to/data/bob.json.gz", ...]
# doctest: +SKIP
```

**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`:

```python
>>> b_dict.map(json.dumps).to_textfiles("/path/to/data/*.json") # doctest: +SKIP
```

**Last endline**: By default the last line does not end with a newline character. Pass `last_endline=True` to invert the default.

### `topk(k, key=None, split_every=None)`
K largest elements in collection

Optionally ordered by some key function

```python
>>> b = from_sequence([10, 3, 5, 7, 11, 4])
>>> list(b.topk(2)) # doctest: +SKIP
[11, 10]
```

```python
>>> list(b.topk(2, lambda x: -x)) # doctest: +SKIP
[3, 4]
```

### `unzip(n)`
Transform a bag of tuples to n bags of their elements.

**Examples**

```python
>>> 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:
```python
>>> first, second, third = (b.pluck(i) for i in range(3))
```

```python
var(ddof=0)
```

### 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

```python
>>> 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

```python
>>> x, y, z = [delayed(load_sequence_from_file)(fn) for fn in filenames] # doctest: +SKIP

>>> b = from_delayed([x, y, z]) # doctest: +SKIP
```
dask.bag.read_text (urlpath, blocksize=None, compression='infer', encoding='utf-8', errors='strict', linedelimiter='
', collection=True, storage_options=None, files_per_partition=None)

Read lines from text files

Parameters

urlpath  [string or list] Absolute or relative filepath(s). Prefix with a protocol like s3:// to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol.

blocksize: None, int, or str  Size (in bytes) to cut up larger files. Streams by default. Can be None for streaming, an integer number of bytes, or a string like “128MiB”

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.

files_per_partition: None or int  If set, group input files into partitions of the requested size, instead of one partition per file. Mutually exclusive with blocksize.

Returns

dask.bag.Bag if collection is True or list of Delayed lists otherwise

See also:

from_sequence  Build bag from Python sequence

Examples

```python
>>> b = read_text('myfiles.1.txt')  # doctest: +SKIP
>>> b = read_text('myfiles.*.txt')  # doctest: +SKIP
>>> b = read_text('myfiles.*.txt.gz')  # doctest: +SKIP
>>> b = read_text('s3://bucket/myfiles.*.txt')  # doctest: +SKIP
>>> b = read_text('s3://key:secret@bucket/myfiles.*.txt')  # doctest: +SKIP
>>> b = read_text('hdfs://namenode.example.com/myfiles.*.txt')  # doctest: +SKIP
```

Parallelize a large file by providing the number of uncompressed bytes to load into each partition.

```python
>>> b = read_text('largefile.txt', blocksize='10MB')  # doctest: +SKIP
```

dask.bag.from_url (urls)

Create a dask Bag from a url.

Examples
```python
>>> a = from_url('http://raw.githubusercontent.com/dask/dask/master/README.rst')
  # doctest: +SKIP
>>> a.npartitions  # doctest: +SKIP
1

>>> a.take(8)  # doctest: +SKIP
(b'Dask
',
  b'====
',
  b'
',
  b'|Build Status| |Coverage| |Doc Status| |Gitter| |Version Status|
',
  b'
',
  b'Dask is a flexible parallel computing library for analytics. See
',
  b'documentation_ for more information.
',
  b'
')

>>> b = from_url(['http://github.com', 'http://google.com'])
  # doctest: +SKIP
>>> b.npartitions  # doctest: +SKIP
2

```

**dask.bag.read_avro**

Read set of avro files

Use this with arbitrary nested avro schemas. Please refer to the fastavro documentation for its capabilities: https://github.com/fastavro/fastavro

**Parameters**

- **urlpath**: string or list
  - Absolute or relative filepath, URL (may include protocols like s3:/), or globstring pointing to data.
- **blocksize**: int or None
  - Size of chunks in bytes. If None, there will be no chunking and each file will become one partition.
- **storage_options**: dict or None
  - passed to backend file-system
- **compression**: str or None
  - Compression format of the target(s), like ‘gzip’. Should only be used with blocksize=None.

**dask.bag.range**

Numbers from zero to n

**Examples**

```python
>>> import dask.bag as db
>>> b = db.range(5, npartitions=2)
>>> list(b)
[0, 1, 2, 3, 4]
```

**dask.bag.concat**

Concatenate many bags together, unioning all elements.

```python
>>> 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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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/kwars 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

```python
>>> 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:

```python
db.map(lambda x: x + 1, b).compute()
```

```
[1, 2, 3, 4, 5]
```

Apply a function with arguments from multiple bags:

```python
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:

```python
db.map(add, b, 1).compute()
```

```
[1, 2, 3, 4, 5]
```

Keyword arguments are also supported, and have the same semantics as regular arguments:

```python
def myadd(x, y=0):
    return x + y

db.map(myadd, b, y=b2).compute()
```

```
[5, 7, 9, 11, 13]
```

```python
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:

```python
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:

```python
>>> 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:

```python
>>> numbers = db.range(20) # doctest: +SKIP
>>> fizz = numbers.filter(lambda n: n % 3 == 0) # doctest: +SKIP
>>> buzz = numbers.filter(lambda n: n % 5 == 0) # doctest: +SKIP
>>> fizzbuzz = db.zip(fizz, buzz) # doctest: +SKIP
>>> list(fizzbuzzzz) # doctest: +SKIP
[(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:

```python
>>> list(fizzbuzzz) # doctest: +SKIP
[(0, 0), (3, None), (None, 5), (6, None), (None 10), (9, None),
(12, None), (15, 15), (18, None), (None, 20), (None, 25), (None, 30)]
```

## 4.9 DataFrame

### 4.9.1 API

**DataFrame**

- **DataFrame(dsk, name, meta, divisions)**: Parallel Pandas DataFrame
- **DataFrame.add(other[, axis, level, fill_value])**: Addition of dataframe and other, element-wise (binary operator `add`).
- **DataFrame.append(other[, interleave_partitions])**: Append rows of `other` to the end of caller, returning a new object.
- **DataFrame.apply(func[, axis, broadcast, . . .])**: Parallel version of pandas.DataFrame.apply
- **DataFrame.assign(**kwargs)**: Assign new columns to a DataFrame.
- **DataFrame.astype(dtype)**: Cast a pandas object to a specified dtype `dtype`.
- **DataFrame.categorize([columns, index, . . .])**: Convert columns of the DataFrame to category dtype.
- **DataFrame.columns**
- **DataFrame.compute(**kwargs)**: Compute this dask collection
- **DataFrame.corr([method, min_periods, . . .])**: Compute pairwise correlation of columns, excluding NA/null values.
- **DataFrame.count([axis, split_every])**: Count non-NA cells for each column or row.
- **DataFrame.cov([min_periods, split_every])**: Compute pairwise covariance of columns, excluding NA/null values.
- **DataFrame.cummax([axis, skipna, out])**: Return cumulative maximum over a DataFrame or Series axis.
- **DataFrame.cummin([axis, skipna, out])**: Return cumulative minimum over a DataFrame or Series axis.
- **DataFrame.cumprod([axis, skipna, dtype, out])**: Return cumulative product over a DataFrame or Series axis.
- **DataFrame.cumsum([axis, skipna, dtype, out])**: Return cumulative sum over a DataFrame or Series axis.
- **DataFrame.describe([split_every, . . .])**: Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.
- **DataFrame.div(other[, axis, level, fill_value])**: Floating division of dataframe and other, element-wise (binary operator `truediv`).
- **DataFrame.drop(labels[, axis, errors])**: Drop specified labels from rows or columns.
### Table 33 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.drop_duplicates([split_every, ...])</code></td>
<td>Return DataFrame with duplicate rows removed, optionally only considering certain columns.</td>
</tr>
<tr>
<td><code>DataFrame.dropna([how, subset, thresh])</code></td>
<td>Remove missing values.</td>
</tr>
<tr>
<td><code>DataFrame.dtypes</code></td>
<td>Return data types</td>
</tr>
<tr>
<td><code>DataFrame.fillna([value, method, limit, axis])</code></td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>DataFrame.floordiv(other[, axis, level, ...])</code></td>
<td>Integer division of dataframe and other, element-wise (binary operator <code>floordiv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.get_partition(n)</code></td>
<td>Get a dask DataFrame/Series representing the <code>n</code>th partition.</td>
</tr>
<tr>
<td><code>DataFrame.groupby([by])</code></td>
<td>Group DataFrame or Series using a mapper or by a Series of columns.</td>
</tr>
<tr>
<td><code>DataFrame.head([n, npartitions, compute])</code></td>
<td>First <code>n</code> rows of the dataset</td>
</tr>
<tr>
<td><code>DataFrame.iloc</code></td>
<td>Purely integer-location based indexing for selection by position.</td>
</tr>
<tr>
<td><code>DataFrame.index</code></td>
<td>Return dask Index instance</td>
</tr>
<tr>
<td><code>DataFrame.isna()</code></td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>DataFrame.isnull()</code></td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>DataFrame.iterrows()</code></td>
<td>Iterate over DataFrame rows as (index, Series) pairs.</td>
</tr>
<tr>
<td><code>DataFrame.itertuples([index, name])</code></td>
<td>Iterate over DataFrame rows as namedtuples.</td>
</tr>
<tr>
<td><code>DataFrame.join(other[, on, how, lsuffix, ...])</code></td>
<td>Join columns of another DataFrame</td>
</tr>
<tr>
<td><code>DataFrame.Known_divisions</code></td>
<td>Whether divisions are already known</td>
</tr>
<tr>
<td><code>DataFrame.loc</code></td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td><code>DataFrame.map_partitions(func, *args, **kwangs)</code></td>
<td>Apply Python function on each DataFrame partition.</td>
</tr>
<tr>
<td><code>DataFrame.mask(cond[, other])</code></td>
<td>Replace values where the condition is True.</td>
</tr>
<tr>
<td><code>DataFrame.max([axis, skipna, split_every, out])</code></td>
<td>Return the maximum of the values for the requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.mean([axis, skipna, split_every, ...])</code></td>
<td>Return the mean of the values for the requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.merge(right[, how, on, left_on, ...])</code></td>
<td>Merge the DataFrame with another DataFrame</td>
</tr>
<tr>
<td><code>DataFrame.min([axis, skipna, split_every, out])</code></td>
<td>Return the minimum of the values for the requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.mod(other[, axis, level, fill_value])</code></td>
<td>Modulo of dataframe and other, element-wise (binary operator <code>mod</code>).</td>
</tr>
<tr>
<td><code>DataFrame.mul(other[, axis, level, fill_value])</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>DataFrame.ndim</code></td>
<td>Return dimensionality</td>
</tr>
<tr>
<td><code>DataFrame.nlargest([n, columns, split_every])</code></td>
<td>Return the first <code>n</code> rows ordered by <code>columns</code> in descending order.</td>
</tr>
<tr>
<td><code>DataFrame.npartitions</code></td>
<td>Return number of partitions</td>
</tr>
<tr>
<td><code>DataFrame.partitions</code></td>
<td>Slice dataframe by partitions</td>
</tr>
<tr>
<td><code>DataFrame.pow(other[, axis, level, fill_value])</code></td>
<td>Exponential power of dataframe and other, element-wise (binary operator <code>pow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.prod([axis, skipna, split_every, ...])</code></td>
<td>Return the product of the values for the requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.quantile([q, axis, method])</code></td>
<td>Approximate row-wise and precise column-wise quantiles of DataFrame</td>
</tr>
<tr>
<td><code>DataFrame.query(expr, **kwargs)</code></td>
<td>Filter dataframe with complex expression</td>
</tr>
<tr>
<td><code>DataFrame.radd(other[, axis, level, fill_value])</code></td>
<td>Addition of dataframe and other, element-wise (binary operator <code>radd</code>).</td>
</tr>
<tr>
<td><code>DataFrame.random_split(frac[, random_state])</code></td>
<td>Pseudorandomly split dataframe into different pieces row-wise</td>
</tr>
</tbody>
</table>

Continued on next page
### DataFrame

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.rdiv(other[, axis, level, fill_value])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rename(index, columns)</code></td>
<td>Alter axes labels.</td>
</tr>
<tr>
<td><code>DataFrame.repartition([divisions, ...])</code></td>
<td>Repartition dataframe along new divisions.</td>
</tr>
<tr>
<td><code>DataFrame.replace([to_replace, value, regex])</code></td>
<td>Replace values given in <code>to_replace</code> with <code>value</code>.</td>
</tr>
<tr>
<td><code>DataFrame.reset_index([drop])</code></td>
<td>Reset the index to the default index.</td>
</tr>
<tr>
<td><code>DataFrame.rfloordiv(other[, axis, level, ...])</code></td>
<td>Integer division of dataframe and other, element-wise (binary operator <code>rfloordiv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rmod(other[, axis, level, fill_value])</code></td>
<td>Modulo of dataframe and other, element-wise (binary operator <code>rmod</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rfmul(other[, axis, level, fill_value])</code></td>
<td>Multiplication of dataframe and other, element-wise (binary operator <code>rmul</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rpow(other[, axis, level, fill_value])</code></td>
<td>Exponential power of dataframe and other, element-wise (binary operator <code>rpow</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rsub(other[, axis, level, fill_value])</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator <code>rsub</code>).</td>
</tr>
<tr>
<td><code>DataFrame.rtruediv(other[, axis, level, ...])</code></td>
<td>Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code>).</td>
</tr>
<tr>
<td><code>DataFrame.shape</code></td>
<td>Return a tuple representing the dimensionality of the DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.std([axis, skipna, ddof, ...])</code></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.sub(other[, axis, level, fill_value])</code></td>
<td>Subtraction of dataframe and other, element-wise (binary operator <code>sub</code>).</td>
</tr>
<tr>
<td><code>DataFrame.sum([axis, skipna, split_every, ...])</code></td>
<td>Return the sum of the values for the requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.tail([n, compute])</code></td>
<td>Last <code>n</code> rows of the dataset.</td>
</tr>
<tr>
<td><code>DataFrame.to_bag([index])</code></td>
<td>Create Dask Bag from a Dask DataFrame.</td>
</tr>
<tr>
<td><code>DataFrame.to_csv(filename, **kwargs)</code></td>
<td>Store Dask DataFrame to CSV files.</td>
</tr>
<tr>
<td><code>DataFrame.to_dask_array([lengths])</code></td>
<td>Convert a dask DataFrame to a dask array.</td>
</tr>
<tr>
<td><code>DataFrame.to_delayed([optimize_graph])</code></td>
<td>Convert into a list of <code>dask.delayed</code> objects, one per partition.</td>
</tr>
<tr>
<td><code>DataFrame.to_hdf(path_or_buf, key[, mode, ...])</code></td>
<td>Store Dask DataFrame to Hierarchical Data Format (HDF) files.</td>
</tr>
<tr>
<td><code>DataFrame.to_json(filename, *args, **kwargs)</code></td>
<td>See <code>dd.to_json</code> docstring for more information.</td>
</tr>
<tr>
<td><code>DataFrame.to_parquet(path, *args, **kwargs)</code></td>
<td>Store Dask.dataframe to Parquet files.</td>
</tr>
<tr>
<td><code>DataFrame.values</code></td>
<td>Return a dask.array of the values of this dataframe.</td>
</tr>
<tr>
<td><code>DataFrame.var([axis, skipna, ddof, ...])</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td><code>DataFrame.visualize([filename, format, ...])</code></td>
<td>Render the computation of this object’s task graph using graphviz.</td>
</tr>
<tr>
<td><code>DataFrame.where(cond[, other])</code></td>
<td>Replace values where the condition is False.</td>
</tr>
</tbody>
</table>

### Series

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series(dsk, name, meta, divisions)</code></td>
<td>Parallel Pandas Series</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Series.add</strong>(other[, level, fill_value, axis])</td>
<td>Addition of series and other, element-wise (binary operator <code>add</code>).</td>
</tr>
<tr>
<td><strong>Series.align</strong>(other[, join, axis, fill_value])</td>
<td>Align two objects on their axes with the specified join method for each axis Index.</td>
</tr>
<tr>
<td><strong>Series.all</strong>(axis, skipna, split_every, out)</td>
<td>Return whether all elements are True, potentially over an axis.</td>
</tr>
<tr>
<td><strong>Series.any</strong>(axis, skipna, split_every, out)</td>
<td>Return whether any element is True, potentially over an axis.</td>
</tr>
<tr>
<td><strong>Series.append</strong>(other[, interleave_partitions])</td>
<td>Concatenate two or more Series.</td>
</tr>
<tr>
<td><strong>Series.apply</strong>(func[, convert_dtype, meta, args])</td>
<td>Parallel version of pandas.Series.apply</td>
</tr>
<tr>
<td><strong>Series.astype</strong>(dtype)</td>
<td>Cast a pandas object to a specified dtype <code>dtype</code>.</td>
</tr>
<tr>
<td><strong>Series.autocorr</strong>(lag, split_every)</td>
<td>Compute the lag-N autocorrelation.</td>
</tr>
<tr>
<td><strong>Series.between</strong>(left, right[, inclusive])</td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right.</td>
</tr>
<tr>
<td><strong>Series.bfill</strong>(axis, limit)</td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='bfill'.</td>
</tr>
<tr>
<td><strong>Series.clear_divisions()</strong></td>
<td>Forget division information</td>
</tr>
<tr>
<td><strong>Series.clip</strong>(lower, upper, out)</td>
<td>Trim values at input threshold(s).</td>
</tr>
<tr>
<td><strong>Series.clip_lower</strong>(threshold)</td>
<td>Trim values below a given threshold.</td>
</tr>
<tr>
<td><strong>Series.clip_upper</strong>(threshold)</td>
<td>Trim values above a given threshold.</td>
</tr>
<tr>
<td><strong>Series.compute</strong>(**kwargs)</td>
<td>Compute this dask collection</td>
</tr>
<tr>
<td><strong>Series.copy()</strong></td>
<td>Make a copy of the dataframe</td>
</tr>
<tr>
<td><strong>Series.corr</strong>(other[, method, min_periods, ...])</td>
<td>Compute correlation with <code>other</code> Series, excluding missing values.</td>
</tr>
<tr>
<td><strong>Series.count</strong>(split_every)</td>
<td>Return number of non-NA/null observations in the Series.</td>
</tr>
<tr>
<td><strong>Series.cov</strong>(other[, min_periods, split_every])</td>
<td>Compute covariance with Series, excluding missing values.</td>
</tr>
<tr>
<td><strong>Series.cummax</strong>(axis, skipna, out)</td>
<td>Return cumulative maximum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.cummin</strong>(axis, skipna, out)</td>
<td>Return cumulative minimum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.cumprod</strong>(axis, skipna, dtype, out)</td>
<td>Return cumulative product over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.cumsum</strong>(axis, skipna, dtype, out)</td>
<td>Return cumulative sum over a DataFrame or Series axis.</td>
</tr>
<tr>
<td><strong>Series.describe</strong>(split_every, percentiles, ...)</td>
<td>Generate descriptive statistics that summarize the central tendency, dispersions and shape of a dataset’s distribution, excluding NaN values.</td>
</tr>
<tr>
<td><strong>Series.diff</strong>(periods, axis)</td>
<td>First discrete difference of element.</td>
</tr>
<tr>
<td><strong>Series.div</strong>(other[, level, fill_value, axis])</td>
<td>Floating division of series and other, element-wise (binary operator <code>true</code>div).</td>
</tr>
<tr>
<td><strong>Series.drop_duplicates</strong>(split_every, split_out)</td>
<td>Return DataFrame with duplicate rows removed, optionally only considering certain columns.</td>
</tr>
<tr>
<td><strong>Series.dropna()</strong></td>
<td>Return a new Series with missing values removed.</td>
</tr>
<tr>
<td><strong>Series.dt</strong></td>
<td>Namespace of datetime methods</td>
</tr>
<tr>
<td><strong>Series.dtype</strong></td>
<td>Return data type</td>
</tr>
<tr>
<td><strong>Series.eq</strong>(other[, level, fill_value, axis])</td>
<td>Equal to of series and other, element-wise (binary operator <code>eq</code>).</td>
</tr>
<tr>
<td><strong>Series.ffill</strong>(axis, limit)</td>
<td>Synonym for <code>DataFrame.fillna()</code> with method='ffill'.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 34 — continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.fillna</code>([value, method, limit, axis])</td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>Series.first</code>(offset)</td>
<td>Convenience method for subsetting initial periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><code>Series.floordiv</code>(other[, level, fill_value, axis])</td>
<td>Integer division of series and other, element-wise (binary operator <code>floordiv</code>).</td>
</tr>
<tr>
<td><code>Series.ge</code>(other[, level, fill_value, axis])</td>
<td>Greater than or equal to of series and other, element-wise (binary operator <code>ge</code>).</td>
</tr>
<tr>
<td><code>Series.get_partition</code>(n)</td>
<td>Get a dask DataFrame/Series representing the ( n )th partition.</td>
</tr>
<tr>
<td><code>Series.groupby</code>([by])</td>
<td>Group DataFrame or Series using a mapper or by a Series of columns.</td>
</tr>
<tr>
<td><code>Series.gt</code>(other[, level, fill_value, axis])</td>
<td>Greater than of series and other, element-wise (binary operator <code>gt</code>).</td>
</tr>
<tr>
<td><code>Series.head</code>(n[, npartitions, compute])</td>
<td>First ( n ) rows of the dataset</td>
</tr>
<tr>
<td><code>Series.idxmax</code>([axis, skipna, split_every])</td>
<td>Return index of first occurrence of maximum over requested axis.</td>
</tr>
<tr>
<td><code>Series.idxmin</code>([axis, skipna, split_every])</td>
<td>Return index of first occurrence of minimum over requested axis.</td>
</tr>
<tr>
<td><code>Series.isin</code>(values)</td>
<td>Check whether ( values ) are contained in Series.</td>
</tr>
<tr>
<td><code>Series.isna</code>()</td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>Series.isnull</code>()</td>
<td>Detect missing values.</td>
</tr>
<tr>
<td><code>Series.iteritems</code>()</td>
<td>Lazily iterate over (index, value) tuples.</td>
</tr>
<tr>
<td><code>Series.known_divisions</code></td>
<td>Whether divisions are already known</td>
</tr>
<tr>
<td><code>Series.last</code>(offset)</td>
<td>Convenience method for subsetting final periods of time series data based on a date offset.</td>
</tr>
<tr>
<td><code>Series.le</code>(other[, level, fill_value, axis])</td>
<td>Less than or equal to of series and other, element-wise (binary operator <code>le</code>).</td>
</tr>
<tr>
<td><code>Series.loc</code></td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
<tr>
<td><code>Series.lt</code>(other[, level, fill_value, axis])</td>
<td>Less than of series and other, element-wise (binary operator <code>lt</code>).</td>
</tr>
<tr>
<td><code>Series.map</code>(arg[, na_action, meta])</td>
<td>Map values of Series according to input correspondence.</td>
</tr>
<tr>
<td><code>Series.map_overlap</code>(func, before, after, ... )</td>
<td>Apply a function to each partition, sharing rows with adjacent partitions.</td>
</tr>
<tr>
<td><code>Series.map_partitions</code>(func, *args, **kwargs)</td>
<td>Apply Python function on each DataFrame partition.</td>
</tr>
<tr>
<td><code>Series.mask</code>(cond[, other])</td>
<td>Replace values where the condition is True.</td>
</tr>
<tr>
<td><code>Series.max</code>([axis, skipna, split_every, out])</td>
<td>Return the maximum of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Series.mean</code>([axis, skipna, split_every,...])</td>
<td>Return the mean of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Series.memory_usage</code>([index, deep])</td>
<td>Return the memory usage of the Series.</td>
</tr>
<tr>
<td><code>Series.min</code>([axis, skipna, split_every, out])</td>
<td>Return the minimum of the values for the requested axis.</td>
</tr>
<tr>
<td><code>Series.mod</code>(other[, level, fill_value, axis])</td>
<td>Modulo of series and other, element-wise (binary operator <code>mod</code>).</td>
</tr>
<tr>
<td><code>Series.mul</code>(other[, level, fill_value, axis])</td>
<td>Multiplication of series and other, element-wise (binary operator <code>mul</code>).</td>
</tr>
<tr>
<td><code>Series.nbytes</code></td>
<td>Number of bytes</td>
</tr>
<tr>
<td><code>Series.ndim</code></td>
<td>Return dimensionality</td>
</tr>
<tr>
<td><code>Series.ne</code>(other[, level, fill_value, axis])</td>
<td>Not equal to of series and other, element-wise (binary operator <code>ne</code>).</td>
</tr>
<tr>
<td><code>Series.nlargest</code>([n, split_every])</td>
<td>Return the largest ( n ) elements.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 34 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Series.notnull()</strong></td>
<td>Detect existing (non-missing) values.</td>
</tr>
<tr>
<td><strong>Series.nsmallest()</strong></td>
<td>Return the smallest $n$ elements.</td>
</tr>
<tr>
<td><strong>Series.nunique()</strong></td>
<td>Return number of unique elements in the object.</td>
</tr>
<tr>
<td><strong>Series.nunique_approx</strong></td>
<td>Approximate number of unique rows.</td>
</tr>
<tr>
<td><strong>Series.persist()</strong></td>
<td>Persist this dask collection into memory</td>
</tr>
<tr>
<td><strong>Series.pow()</strong></td>
<td>Exponential power of series and other, element-wise (binary operator $\text{pow}$).</td>
</tr>
<tr>
<td><strong>Series.prod()</strong></td>
<td>Return the product of the values for the requested axis.</td>
</tr>
<tr>
<td><strong>Series.quantile()</strong></td>
<td>Approximate quantiles of Series</td>
</tr>
<tr>
<td><strong>Series.radd()</strong></td>
<td>Addition of series and other, element-wise (binary operator $\text{radd}$).</td>
</tr>
<tr>
<td><strong>Series.random_split()</strong></td>
<td>Pseudorandomly split dataframe into different pieces row-wise</td>
</tr>
<tr>
<td><strong>Series.rdiv()</strong></td>
<td>Floating division of series and other, element-wise (binary operator $\text{rtruediv}$).</td>
</tr>
<tr>
<td><strong>Series.reduction()</strong></td>
<td>Generic row-wise reductions.</td>
</tr>
<tr>
<td><strong>Series.repartition()</strong></td>
<td>Repartition dataframe along new divisions</td>
</tr>
<tr>
<td><strong>Series.replace()</strong></td>
<td>Replace values given in $\text{to_replace}$ with $\text{value}$.</td>
</tr>
<tr>
<td><strong>Series.rename()</strong></td>
<td>Alter Series index labels or name</td>
</tr>
<tr>
<td><strong>Series.resample()</strong></td>
<td>Resample time-series data.</td>
</tr>
<tr>
<td><strong>Series.reset_index()</strong></td>
<td>Reset the index to the default index.</td>
</tr>
<tr>
<td><strong>Series.rolling()</strong></td>
<td>Provides rolling transformations.</td>
</tr>
<tr>
<td><strong>Series.round()</strong></td>
<td>Round each value in a Series to the given number of decimals.</td>
</tr>
<tr>
<td><strong>Series.sample()</strong></td>
<td>Random sample of items</td>
</tr>
<tr>
<td><strong>Series.sem()</strong></td>
<td>Return unbiased standard error of the mean over requested axis.</td>
</tr>
<tr>
<td><strong>Series.shape</strong></td>
<td>Return a tuple representing the dimensionality of a Series.</td>
</tr>
<tr>
<td><strong>Series.shift()</strong></td>
<td>Shift index by desired number of periods with an optional time $freq$.</td>
</tr>
<tr>
<td><strong>Series.size</strong></td>
<td>Size of the Series or DataFrame as a Delayed object.</td>
</tr>
<tr>
<td><strong>Series.std()</strong></td>
<td>Return sample standard deviation over requested axis.</td>
</tr>
<tr>
<td><strong>Series.str</strong></td>
<td>Namespace for string methods</td>
</tr>
<tr>
<td><strong>Series.sub()</strong></td>
<td>Subtraction of series and other, element-wise (binary operator $\text{sub}$).</td>
</tr>
<tr>
<td><strong>Series.sum()</strong></td>
<td>Return the sum of the values for the requested axis.</td>
</tr>
<tr>
<td><strong>Series.to_bag()</strong></td>
<td>Create a Dask Bag from a Series.</td>
</tr>
<tr>
<td><strong>Series.to_csv()</strong></td>
<td>Store Dask DataFrame to CSV files</td>
</tr>
<tr>
<td><strong>Series.to_delayed()</strong></td>
<td>Convert a dask DataFrame to a dask array.</td>
</tr>
<tr>
<td><strong>Series.to_delayed()</strong></td>
<td>Convert into a list of dask.delayed objects, one per partition.</td>
</tr>
<tr>
<td><strong>Series.to_frame()</strong></td>
<td>Convert Series to DataFrame.</td>
</tr>
<tr>
<td><strong>Series.to_hdf()</strong></td>
<td>Store Dask Dataframe to Hierarchical Data Format (HDF) files</td>
</tr>
<tr>
<td><strong>Series.to_string()</strong></td>
<td>Render a string representation of the Series.</td>
</tr>
<tr>
<td><strong>Series.to_timestamp()</strong></td>
<td>Cast to DatetimeIndex of timestamps, at $beginning$ of period.</td>
</tr>
<tr>
<td><strong>Series.truediv()</strong></td>
<td>Floating division of series and other, element-wise (binary operator $\text{truediv}$).</td>
</tr>
</tbody>
</table>

Continued on next page
Table 34 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.unique(split_every, split_out)</code></td>
<td>Return Series of unique values in the object.</td>
</tr>
<tr>
<td><code>Series.value_counts(split_every, split_out)</code></td>
<td>Return a Series containing counts of unique values.</td>
</tr>
<tr>
<td><code>Series.var(axes, skipna, ddof, ...)</code></td>
<td>Return unbiased variance over requested axis.</td>
</tr>
<tr>
<td><code>Series.visualize(filename, format, ...)</code></td>
<td>Render the computation of this object’s task graph using graphviz.</td>
</tr>
<tr>
<td><code>Series.where(cond[, other])</code></td>
<td>Replace values where the condition is False.</td>
</tr>
</tbody>
</table>

**Groupby Operations**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrameGroupBy.aggregate(arg[...])</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.apply(func, *args, **kwargs)</code></td>
<td>Parallel version of pandas GroupBy.apply</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.count(split_every, split_out)</code></td>
<td>Compute count of group, excluding missing values.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.cumcount(axis)</code></td>
<td>Number each item in each group from 0 to the length of that group - 1.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.cumprod(axis)</code></td>
<td>Cumulative product for each group.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.cumsum(axis)</code></td>
<td>Cumulative sum for each group.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.get_group(key)</code></td>
<td>Constructs NDFrame from group with provided name.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.max(split_every, split_out)</code></td>
<td>Compute max of group values See Also —— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.mean(split_every, split_out)</code></td>
<td>Compute mean of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.min(split_every, split_out)</code></td>
<td>Compute min of group values See Also —— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.size(split_every, split_out)</code></td>
<td>Compute group sizes.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.std(ddof, split_every,...)</code></td>
<td>Compute standard deviation of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.sum(split_every,...)</code></td>
<td>Compute sum of group values See Also —— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.var(ddof, split_every,...)</code></td>
<td>Compute variance of groups, excluding missing values.</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.first(split_every, split_out)</code></td>
<td>Compute first of group values See Also —— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby</td>
</tr>
<tr>
<td><code>DataFrameGroupBy.last(split_every, split_out)</code></td>
<td>Compute last of group values See Also —— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby</td>
</tr>
</tbody>
</table>

**SeriesGroupBy**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SeriesGroupBy.aggregate(arg[, split_every, ...])</code></td>
<td>Aggregate using one or more operations over the specified axis.</td>
</tr>
<tr>
<td><code>SeriesGroupBy.apply(func, *args, **kwargs)</code></td>
<td>Parallel version of pandas GroupBy.apply</td>
</tr>
<tr>
<td><code>SeriesGroupBy.count(split_every, split_out)</code></td>
<td>Compute count of group, excluding missing values.</td>
</tr>
<tr>
<td><code>SeriesGroupBy.cumcount(axis)</code></td>
<td>Number each item in each group from 0 to the length of that group - 1.</td>
</tr>
</tbody>
</table>
Table 36 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeriesGroupBy.cumprod</td>
<td>Cumulative product for each group.</td>
</tr>
<tr>
<td>SeriesGroupBy.cumsum</td>
<td>Cumulative sum for each group.</td>
</tr>
<tr>
<td>SeriesGroupBy.get_group</td>
<td>Constructs NDFrame from group with provided name.</td>
</tr>
<tr>
<td>SeriesGroupBy.max</td>
<td>Compute max of group values See Also —— pandas.Series.groupby</td>
</tr>
<tr>
<td></td>
<td>pandas.DataFrame.groupby</td>
</tr>
<tr>
<td>SeriesGroupBy.mean</td>
<td>Compute mean of groups, excluding missing values.</td>
</tr>
<tr>
<td>SeriesGroupBy.min</td>
<td>Compute min of group values See Also —— pandas.Series.groupby</td>
</tr>
<tr>
<td></td>
<td>pandas.DataFrame.groupby</td>
</tr>
<tr>
<td>SeriesGroupBy.nunique</td>
<td>Compute group sizes.</td>
</tr>
<tr>
<td>SeriesGroupBy.size</td>
<td>Compute standard deviation of groups, excluding missing values.</td>
</tr>
<tr>
<td>SeriesGroupBy.std</td>
<td>Compute sum of group values See Also —— pandas.Series.groupby</td>
</tr>
<tr>
<td></td>
<td>pandas.DataFrame.groupby</td>
</tr>
<tr>
<td>SeriesGroupBy.sum</td>
<td>Compute variance of groups, excluding missing values.</td>
</tr>
<tr>
<td>SeriesGroupBy.var</td>
<td>Compute first of group values See Also —— pandas.Series.groupby</td>
</tr>
<tr>
<td></td>
<td>pandas.DataFrame.groupby</td>
</tr>
<tr>
<td>SeriesGroupBy.first</td>
<td>Compute last of group values See Also —— pandas.Series.groupby</td>
</tr>
<tr>
<td></td>
<td>pandas.DataFrame.groupby</td>
</tr>
</tbody>
</table>

Aggregation(name, chunk, agg[, finalize]) User defined groupby-aggregation.

Rolling Operations

rolling.map_overlap(func, df, before, after, ...) Apply a function to each partition, sharing rows with adjacent partitions.

Series.rolling(window[, min_periods, freq, ...]) Provides rolling transformations.

DataFrame.rolling(window[, min_periods, ...]) Provides rolling transformations.

Rolling.apply(func[, args, kwargs]) The rolling function’s apply function.

Rolling.count() The rolling count of any non-NaN observations inside the window.

Rolling.kurt() Calculate unbiased rolling kurtosis.

Rolling.max() Calculate the rolling maximum.

Rolling.mean() Calculate the rolling mean of the values.

Rolling.median() Calculate the rolling median.

Rolling.min() Calculate the rolling minimum.

Rolling.quantile(quantile) Calculate the rolling quantile.

Rolling.skew() Unbiased rolling skewness.

Rolling.std([ddof]) Calculate rolling standard deviation.

Rolling.sum() Calculate rolling sum of given DataFrame or Series.

Rolling.var([ddof]) Calculate unbiased rolling variance.
Create DataFrames

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_csv</code></td>
<td>Read CSV files into a Dask.DataFrame</td>
</tr>
<tr>
<td><code>read_table</code></td>
<td>Read delimited files into a Dask.DataFrame</td>
</tr>
<tr>
<td><code>read_fwf</code></td>
<td>Read fixed-width files into a Dask.DataFrame</td>
</tr>
<tr>
<td><code>read_parquet</code></td>
<td>Read ParquetFile into a Dask DataFrame</td>
</tr>
<tr>
<td><code>read_hdf</code></td>
<td>Read HDF files into a Dask DataFrame</td>
</tr>
<tr>
<td><code>read_json</code></td>
<td>Create a dataframe from a set of JSON files</td>
</tr>
<tr>
<td><code>read_orc</code></td>
<td>Read dataframe from ORC file(s)</td>
</tr>
<tr>
<td><code>read_sql_table</code></td>
<td>Create dataframe from an SQL table.</td>
</tr>
<tr>
<td><code>from_array</code></td>
<td>Read any slicable array into a Dask Dataframe</td>
</tr>
<tr>
<td><code>from_bcolz</code></td>
<td>Read BColz CTable into a Dask Dataframe</td>
</tr>
<tr>
<td><code>from_dask_array</code></td>
<td>Create a Dask DataFrame from a Dask Array.</td>
</tr>
<tr>
<td><code>from_delayed</code></td>
<td>Create Dask DataFrame from many Dask Delayed objects</td>
</tr>
<tr>
<td><code>from_pandas</code></td>
<td>Construct a Dask DataFrame from a Pandas DataFrame</td>
</tr>
<tr>
<td><code>dask.bag.core.Bag.to_dataframe</code></td>
<td>Create Dask Dataframe from a Dask Bag.</td>
</tr>
</tbody>
</table>

Store DataFrames

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_csv</code></td>
<td>Store Dask DataFrame to CSV files</td>
</tr>
<tr>
<td><code>to_parquet</code></td>
<td>Store Dask.dataframe to Parquet files</td>
</tr>
<tr>
<td><code>to_hdf</code></td>
<td>Store Dask Dataframe to Hierarchical Data Format (HDF) files</td>
</tr>
<tr>
<td><code>to_records</code></td>
<td>Create Dask Array from a Dask DataFrame</td>
</tr>
<tr>
<td><code>to_bag</code></td>
<td>Create Dask Bag from a Dask Dataframe</td>
</tr>
<tr>
<td><code>to_json</code></td>
<td>Write dataframe into JSON text files</td>
</tr>
</tbody>
</table>

Convert DataFrames

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_dask_array</code></td>
<td></td>
</tr>
<tr>
<td><code>to_delayed</code></td>
<td></td>
</tr>
</tbody>
</table>

Reshape DataFrames

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_dummies</code></td>
<td>Convert categorical variable into dummy/indicator variables.</td>
</tr>
<tr>
<td><code>pivot_table</code></td>
<td>Create a spreadsheet-style pivot table as a DataFrame.</td>
</tr>
<tr>
<td><code>melt</code></td>
<td></td>
</tr>
</tbody>
</table>

DataFrame Methods

```python
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 Documentation, Release 1.2.2

**dsk:** 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 a Series/DataFrame with absolute numeric value of each element.

This docstring was copied from pandas.core.frame.DataFrame.abs.

Some inconsistencies with the Dask version may exist.

This function only applies to elements that are all numeric.

**Returns**

- **abs**  Series/DataFrame containing the absolute value of each element.

**See also:**

- **numpy.abs**  Calculate the absolute value element-wise.

**Notes**

For complex inputs, \(1.2 + 1j\), the absolute value is \(\sqrt{a^2 + b^2}\).

**Examples**

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])  # doctest: +SKIP
>>> s.abs()  # doctest: +SKIP
    0    1.10
    1     2.0
    2    3.33
    3     4.0
Name: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])  # doctest: +SKIP
>>> s.abs()  # doctest: +SKIP
    0    1.56205
Name: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])  # doctest: +SKIP
>>> s.abs()  # doctest: +SKIP
    0    1 days
Name: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).
>>> df = pd.DataFrame({'a': [4, 5, 6, 7],
... 'b': [10, 20, 30, 40],
... 'c': [100, 50, -30, -50]})
>>> df
   a  b  c
0  4 10 100
1  5 20  50
2  6 30 -30
3  7 40 -50

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. With reverse version, radd.
Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters
- other [scalar, sequence, Series, or DataFrame] Any single or multiple element data struc-
ture, or list-like object.
- axis [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or
  columns (1 or ‘columns’). For Series input, axis to match Series index on.
- level [int or label] Broadcast across a level, matching Index values on the passed Multi-
  Index level.
- fill_value [float or None, default None] Fill existing missing (NaN) values, and any new
  element needed for successful DataFrame alignment, with this value before computa-
tion. If data in both corresponding DataFrame locations is missing the result will be
  missing.

Returns
DataFrame Result of the arithmetic operation.

See also:
DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.
Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
                     'degrees': [360, 180, 360]},
                    index=['circle', 'triangle', 'rectangle'])
>>> df
angles  degrees
circle    0       360
triangle   3       180
rectangle  4       360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
angles  degrees
circle    1       361
triangle   4       181
rectangle  5       361
```

```
>>> df.add(1)  # doctest: +SKIP
angles  degrees
circle    1       361
triangle   4       181
rectangle  5       361
```

Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle    0.0    36.0
triangle   0.3    18.0
rectangle  0.4    36.0
```

```python
>>> df.rdiv(10)  # doctest: +SKIP
angles  degrees
circle    inf    0.027778
triangle   3.333333    0.055556
rectangle  2.500000    0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
angles  degrees
circle    -1    358
triangle    2    178
rectangle    3    358
```

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
angles  degrees
circle    -1    358
triangle    2    178
rectangle    3    358
```
Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4],
                        'degrees': [360, 180, 360, 360, 540, 720]},
                       index=['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])
```
Align two objects on their axes with the specified join method for each axis.

This docstring was copied from pandas.core.frame.DataFrame.align.

Some inconsistencies with the Dask version may exist.

**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 (Not supported in Dask)] Broadcast across a level, matching Index values on the passed MultiIndex level
- `copy` [boolean, default True (Not supported in Dask)] 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` [{'backfill', 'bfill', 'pad', 'ffill'}, default None (Not supported in Dask)] 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
- `limit` [int, default None (Not supported in Dask)] 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.
- `fill_axis` [{0 or 'index', 1 or 'columns'}, default 0 (Not supported in Dask)] Filling axis, method and limit
- `broadcast_axis` [{0 or 'index', 1 or 'columns'}, default None (Not supported in Dask)] Broadcast values along this axis, if aligning two objects of different dimensions

**Returns**

- `(left, right)` [(DataFrame, type of other)] Aligned objects

-all (axis=None, skipna=True, split_every=False, out=None)

Return whether all elements are True, potentially over an axis.

This docstring was copied from pandas.core.frame.DataFrame.all.

Some inconsistencies with the Dask version may exist.

Returns True unless there at least one element within a series or along a DataFrame axis that is False or equivalent (e.g. zero or empty).

**Parameters**

- `axis` [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.
  - 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
  - 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
• None : reduce all axes, return a scalar.

**bool_only** [bool, default None (Not supported in Dask)] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**skipna** [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

**level** [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

**kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

**Series or DataFrame** If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

**Series.all** Return True if all elements are True.

**DataFrame.any** Return True if one (or more) elements are True.

Examples

Series

```python
>>> pd.Series([True, True]).all() # doctest: +SKIP
True
>>> pd.Series([True, False]).all() # doctest: +SKIP
False
>>> pd.Series([]).all() # doctest: +SKIP
True
>>> pd.Series([np.nan]).all() # doctest: +SKIP
True
>>> pd.Series([np.nan]).all(skipna=False) # doctest: +SKIP
True
```

DataFrames

Create a dataframe from a dictionary.

```python
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
```

Default behaviour checks if column-wise values all return True.

```python
>>> df.all() # doctest: +SKIP
coll  col2
0    True  True
1    True  False
dtype: bool
```

---

4.9. DataFrame
Specify `axis='columns'` to check if row-wise values all return True.

```python
>>> df.all(axis='columns')  # doctest: +SKIP
0    True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```python
>>> df.all(axis=None)  # doctest: +SKIP
False
```

The function `any` is used to check if any element is True, potentially over an axis.

```python
any (axis=None, skipna=True, split_every=False, out=None)
```

This docstring was copied from pandas.core.frame.DataFrame.any.

Some inconsistencies with the Dask version may exist.

Returns False unless there at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

**Parameters**

- **axis** [{0 or ‘index’, 1 or ‘columns’, None}, default 0] Indicate which axis or axes should be reduced.
  - 0 / ‘index’ : reduce the index, return a Series whose index is the original column labels.
  - 1 / ‘columns’ : reduce the columns, return a Series whose index is the original index.
  - None : reduce all axes, return a scalar.

- **bool_only** [bool, default None (Not supported in Dask)] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

- **skipna** [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

- **level** [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

- ****kwargs [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- Series or DataFrame If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

- `numpy.any` Numpy version of this method.
- `Series.any` Return whether any element is True.
- `Series.all` Return whether all elements are True.
- `DataFrame.any` Return whether any element is True over requested axis.
dask Documentation, Release 1.2.2

DataFrame.all Return whether all elements are True over requested axis.
Examples
Series
For Series input, the output is a scalar indicating whether any element is True.
>>> pd.Series([False, False]).any() # doctest: +SKIP
False
>>> pd.Series([True, False]).any() # doctest: +SKIP
True
>>> pd.Series([]).any() # doctest: +SKIP
False
>>> pd.Series([np.nan]).any() # doctest: +SKIP
False
>>> pd.Series([np.nan]).any(skipna=False) # doctest: +SKIP
True

DataFrame
Whether each column contains at least one True element (the default).
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
˓→+SKIP
>>> df # doctest: +SKIP
A B C
0 1 0 0
1 2 2 0
>>> df.any()
A
True
B
True
C
False
dtype: bool

# doctest:

# doctest: +SKIP

Aggregating over the columns.
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df # doctest: +SKIP
A B
0
True 1
1 False 2
>>> df.any(axis='columns')
0
True
1
True
dtype: bool

# doctest: +SKIP

>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df # doctest: +SKIP
A B
0
True 1
1 False 0

4.9. DataFrame

# doctest: +SKIP

# doctest: +SKIP

409


Aggregating over the entire DataFrame with axis=None.

any for an empty DataFrame is an empty Series.

append (other, interleave_partitions=False)

Append rows of other to the end of caller, returning a new object.

This docstring was copied from pandas.core.frame.DataFrame.append.

Some inconsistencies with the Dask version may exist.

Columns in other that are not in the caller 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 (Not supported in Dask)] If True, do not use the index labels.
verify_integrity [boolean, default False (Not supported in Dask)] If True, raise ValueError on creating index with duplicates.
sort [boolean, default None (Not supported in Dask)] Sort columns if the columns of self and other are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass sort=True to silence the warning and sort. Explicitly pass sort=False to silence the warning and not sort.

New in version 0.23.0.

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

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))  # doctest: +SKIP
>>> df
   A B
0 1 2
1 3 4

>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))  # doctest: +SKIP
>>> df.append(df2)  # doctest: +SKIP
   A B
0 1 2
1 3 4
0 5 6
1 7 8

With `ignore_index` set to True:

```python
>>> df.append(df2, ignore_index=True)  # doctest: +SKIP
   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:

```python
>>> df = pd.DataFrame(columns=['A'])  # doctest: +SKIP
>>> for i in range(5):  # doctest: +SKIP
...     df = df.append({'A': i}, ignore_index=True)
```

```python
>>> df
   A
0 0
1 1
2 2
3 3
4 4
```

More efficient:

```python
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],  # doctest: +SKIP
...     ignore_index=True)
   A
0 0
1 1
2 2
3 3
4 4
```

```
apply(func, axis=0, broadcast=None, raw=False, reduce=None, args=(), meta='__no_default__', **kwds)
```

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

```python
>>> 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`:

```python
>>> 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:

```python
>>> 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 elementwise.

This docstring was copied from pandas.core.frame.DataFrame.applymap.

Some inconsistencies with the Dask version may exist.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

Parameters

func [callable] Python function, returns a single value from a single value.

Returns

DataFrame Transformed DataFrame.

See also:

DataFrame.apply Apply a function along input axis of DataFrame.

Notes

In the current implementation applymap calls func twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if func has side-effects, as they will take effect twice for the first column/row.

Examples

```python
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
# doctest: +SKIP
>>> df
0   1  
0  1  2.120
1  3.356 4.567

>>> df.applymap(lambda x: len(str(x)))
# doctest: +SKIP
0   1  
0   3  4
1   5  5

Note that a vectorized version of func often exists, which will be much faster. You could square each number elementwise.

```python
>>> df.applymap(lambda x: x**2)
# doctest: +SKIP
0   1  
0  1.000000 4.494400
1  11.262736 20.857489

But it’s better to avoid applymap in that case.

```python
>>> df ** 2
# doctest: +SKIP
0   1  
0  1.000000 4.494400
1  11.262736 20.857489
```
**assign** (**kwargs)

Assign new columns to a DataFrame.

This docstring was copied from pandas.core.frame.DataFrame.assign.

Some inconsistencies with the Dask version may exist.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

**Parameters**

**kwargs [dict of {str: callable or Series}]** The column names are keywords. 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**

DataFrame A new DataFrame with the new columns in addition to all the existing columns.

**Notes**

Assigning multiple columns within the same assign is possible. For Python 3.6 and above, later items in '**kwargs’ may refer to newly created or modified columns in ‘df’; items are computed and assigned into ‘df’ in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

**Examples**

```python
>>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},  # doctest: +SKIP
... index=['Portland', 'Berkeley'])
>>> df               # doctest: +SKIP
     temp_c
Portland  17.0
Berkeley  25.0

Where the value is a callable, evaluated on df:

```python
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)  # doctest: +SKIP
     temp_c    temp_f
Portland  17.0      62.6
Berkeley  25.0      77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```python
>>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)  # doctest: +SKIP
     temp_c    temp_f
Portland  17.0      62.6
Berkeley  25.0      77.0
```

In Python 3.6+, you can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:
astype (dtype)
Cast a pandas object to a specified dtype dtype.

This docstring was copied from pandas.core.frame.DataFrame.astype.

Some inconsistencies with the Dask version may exist.

**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 (Not supported in Dask)] 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' (Not supported in Dask)] 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.

- **kwargs** [keyword arguments to pass on to the constructor]

**Returns**

- **casted** [same type as caller]

See also:

- **to_datetime** Convert argument to datetime.
- **to_timedelta** Convert argument to timedelta.
- **to_numeric** Convert argument to a numeric type.

**numpy.ndarray.astype** Cast a numpy array to a specified type.

**Examples**

```python
>>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32, # doctest: +SKIP
... temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)

temp_c temp_f temp_k
Portland 17.0 62.6 290.15
Berkeley 25.0 77.0 298.15
```

```
astype (dtype)
```

Convert to categorical type:
Convert to ordered categorical type with custom ordering:

```python
>>> cat_dtype = pd.api.types.CategoricalDtype(
...     categories=[2, 1], ordered=True)
```

```python
>>> ser.astype(cat_dtype)  # doctest: +SKIP
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:

```python
>>> s1 = pd.Series([1,2])  # doctest: +SKIP
>>> s2 = s1.astype('int64', copy=False)  # doctest: +SKIP
>>> s2[0] = 10  # doctest: +SKIP
```

```python
>>> s1
0  10
1  2
dtype: int64
```

### bfill

- **Synonym for** `DataFrame.fillna()` with **method**='bfill'.

### categorize

- **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

- **Trim values at input threshold(s).**

  This docstring was copied from `pandas.core.frame.DataFrame.clip`.

  Some inconsistencies with the Dask version may exist.

  **Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.**

  **Parameters**
**lower** [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

**upper** [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

**axis** [int or string axis name, optional (Not supported in Dask)] Align object with lower and upper along the given axis.

**inplace** [boolean, default False (Not supported in Dask)] Whether to perform the operation in place on the data.

New in version 0.21.0.

*args, **kwargs Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns**

**Series or DataFrame** Same type as calling object with the values outside the clip boundaries replaced

**Examples**

```python
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}  # doctest: +SKIP
```

Clips per column using lower and upper thresholds:

```python
>>> df = pd.DataFrame(data)  # doctest: +SKIP
```

```python
>>> df
```

<table>
<thead>
<tr>
<th></th>
<th>col_0</th>
<th>col_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>-3</td>
<td>-7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-5</td>
</tr>
</tbody>
</table>

```python
>>> df.clip(-4, 6)  # doctest: +SKIP
```

```python
>>> df
```

<table>
<thead>
<tr>
<th></th>
<th>col_0</th>
<th>col_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-4</td>
</tr>
</tbody>
</table>

Clips using specific lower and upper thresholds per column element:

```python
t = pd.Series([2, -4, -1, 6, 3])  # doctest: +SKIP
```

```python
t
```

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
```

dtype: int64

```python
>>> df.clip(t, t + 4, axis=0)  # doctest: +SKIP
```

```
<table>
<thead>
<tr>
<th></th>
<th>col_0</th>
<th>col_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-4</td>
</tr>
</tbody>
</table>
```

(continues on next page)
clip_lower \( (\text{threshold}) \)

Trim values below a given threshold.

This docstring was copied from pandas.core.frame.DataFrame.clip_lower.

Some inconsistencies with the Dask version may exist.

Deprecated since version 0.24.0: Use clip(lower=threshold) instead.

Elements below the \textit{threshold} will be changed to match the \textit{threshold} value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

**Parameters**

\textbf{threshold} [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to \textit{threshold}.
- array-like : The shape of \textit{threshold} should match the object it’s compared to. When \textit{self} is a Series, \textit{threshold} should be the length. When \textit{self} is a DataFrame, \textit{threshold} should 2-D and the same shape as \textit{self} for \texttt{axis=None}, or 1-D and the same length as the axis being compared.

\textbf{axis} [(0 or ‘index’, 1 or ‘columns’), default 0 (Not supported in Dask)] Align \textit{self} with \textit{threshold} along the given axis.

\textbf{inplace} [boolean, default False (Not supported in Dask)] Whether to perform the operation in place on the data.

New in version 0.21.0.

**Returns**

\textbf{Series or DataFrame} Original data with values trimmed.

**See also:**

\textit{Series.clip} General purpose method to trim Series values to given threshold(s).

\textit{DataFrame.clip} General purpose method to trim DataFrame values to given threshold(s).

**Examples**

Series single threshold clipping:

```python
>>> s = pd.Series([5, 6, 7, 8, 9]) # doctest: +SKIP
>>> s.clip(lower=8) # doctest: +SKIP
0    8
1    8
2    8
3    8
4    9
dtype: int64
```
Series clipping element-wise using an array of thresholds. `threshold` should be the same length as the Series.

```python
>>> elemwise_thresholds = [4, 8, 7, 2, 5]  # doctest: +SKIP
gsio
>>> s.clip(lower=elemwise_thresholds)  # doctest: +SKIP
0 5
1 8
2 7
3 8
4 9
dtype: int64
```

DataFrames can be compared to a scalar.

```python
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})  # doctest: +SKIP
>>> df  # doctest: +SKIP
     A  B
0  1  2
1  3  4
2  5  6
```

```python
>>> df.clip(lower=3)  # doctest: +SKIP
     A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, `threshold` should be the same shape as the DataFrame.

```python
>>> df.clip(lower=np.array([[3, 4], [2, 2], [6, 2]]))  # doctest: +SKIP
     A  B
0  3  4
1  3  4
2  6  6
```

Control how `threshold` is broadcast with `axis`. In this case `threshold` should be the same length as the axis specified by `axis`.

```python
>>> df.clip(lower=[3, 3, 5], axis='index')  # doctest: +SKIP
     A  B
0  3  3
1  3  4
2  5  6
```

```python
>>> df.clip(lower=[4, 5], axis='columns')  # doctest: +SKIP
     A  B
0  4  5
1  4  5
2  5  6
```

### clip_upper(threshold)

Trim values above a given threshold.

This docstring was copied from pandas.core.frame.DataFrame.clip_upper.

Some inconsistencies with the Dask version may exist.

Deprecated since version 0.24.0: Use clip(upper=threshold) instead.
Elements above the \textit{threshold} will be changed to match the \textit{threshold} value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

\textbf{Parameters}

\textbf{threshold} [numeric or array-like] Maximum value allowed. All values above \textit{threshold} will be set to this value.

\begin{itemize}
\item float : every value is compared to \textit{threshold}.
\item array-like : The shape of \textit{threshold} should match the object it’s compared to. When \textit{self} is a Series, \textit{threshold} should be the length. When \textit{self} is a DataFrame, \textit{threshold} should 2-D and the same shape as \textit{self} for \texttt{axis=None}, or 1-D and the same length as the axis being compared.
\end{itemize}

\textbf{axis} [[0 or \textquoteleft index\textquoteright, 1 or \textquoteleft columns\textquoteright, default 0 (Not supported in Dask)] Align object with \textit{threshold} along the given axis.

\textbf{inplace} [boolean, default False (Not supported in Dask)] Whether to perform the operation in place on the data.

New in version 0.21.0.

\textbf{Returns}

Series or DataFrame Original data with values trimmed.

\textbf{See also:}

\texttt{Series.clip} General purpose method to trim Series values to given threshold(s).

\texttt{DataFrame.clip} General purpose method to trim DataFrame values to given threshold(s).

\textbf{Examples}

\begin{verbatim}
>>> s = pd.Series([1, 2, 3, 4, 5])  # doctest: +SKIP
>>> s # doctest: +SKIP
0   1
1   2
2   3
3   4
4   5
dtype: int64

>>> s.clip(upper=3)  # doctest: +SKIP
0   1
1   2
2   3
3   3
4   3
dtype: int64

>>> elemwise_thresholds = [5, 4, 3, 2, 1]  # doctest: +SKIP
>>> elemwise_thresholds  # doctest: +SKIP
[5, 4, 3, 2, 1]

>>> s.clip(upper=elemwise_thresholds)  # doctest: +SKIP
0   1
1   2
2   3
3   3
4   3
(continues on next page)
\end{verbatim}
combine (other, func, fill_value=None, overwrite=True)

Perform column-wise combine with another DataFrame based on a passed function.

This docstring was copied from pandas.core.frame.DataFrame.combine.

Some inconsistencies with the Dask version may exist.

Combines a DataFrame with other DataFrame using func to element-wise combine columns. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

other [DataFrame] The DataFrame to merge column-wise.

func [function] Function that takes two series as inputs and return a Series or a scalar. Used to merge the two dataframes column by columns.

fill_value [scalar value, default None] The value to fill NaNs with prior to passing any column to the merge func.

overwrite [boolean, default True] If True, columns in self that do not exist in other will be overwritten with NaNs.

Returns

result [DataFrame]

See also:

DataFrame.combine_first Combine two DataFrame objects and default to non-null values in frame calling the method.

Examples

Combine using a simple function that chooses the smaller column.

```python
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]}) # doctest: +SKIP
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]}) # doctest: +SKIP
>>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2 # doctest: +SKIP
>>> df1.combine(df2, take_smaller) # doctest: +SKIP
   A  B
0  0  3
1  0  3
```

Example using a true element-wise combine function.

```python
>>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]}) # doctest: +SKIP
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]}) # doctest: +SKIP
>>> df1.combine(df2, np.minimum) # doctest: +SKIP
   A  B
0  1  2
1  0  3
```
Using \textit{fill\_value} fills Nones prior to passing the column to the merge function.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})  # doctest: +SKIP
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})  # doctest: +SKIP
>>> df1.combine(df2, take_smaller, fill_value=-5)  # doctest: +SKIP
  A  B
0  0 -5.0
1  0  4.0
```

However, if the same element in both dataframes is None, that None is preserved

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})  # doctest: +SKIP
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})  # doctest: +SKIP
>>> df1.combine(df2, take_smaller, fill_value=-5)  # doctest: +SKIP
  A  B
0  0  NaN
1  0  3.0
```

Example that demonstrates the use of \textit{overwrite} and behavior when the axis differ between the dataframes.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})  # doctest: +SKIP
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1]}, index=[1, 2])  # doctest: +SKIP
>>> df1.combine(df2, take_smaller)  # doctest: +SKIP
  A  B  C
0  NaN NaN NaN
1  0.0  3.0 -10.0
2  NaN  3.0  1.0
```

```
>>> df1.combine(df2, take_smaller, overwrite=False)  # doctest: +SKIP
  A  B  C
0  0.0 NaN NaN
1  0.0  3.0 -10.0
2  NaN  3.0  1.0
```

Demonstrating the preference of the passed in dataframe.

```
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])  # doctest: +SKIP
>>> df2.combine(df1, take_smaller)  # doctest: +SKIP
  A  B  C
0  NaN NaN NaN
1  0.0  3.0  NaN
2  NaN  3.0  NaN
```

```
>>> df2.combine(df1, take_smaller, overwrite=False)  # doctest: +SKIP
  A  B  C
0  0.0 NaN NaN
1  0.0  3.0  1.0
2  NaN  3.0  1.0
```

\texttt{combine\_first} (\texttt{other})

Update null elements with value in the same location in \texttt{other}.

This docstring was copied from pandas.core.frame.DataFrame.combine\_first.

Some inconsistencies with the Dask version may exist.
Combine two DataFrame objects by filling null values in one DataFrame with non-null values from other DataFrame. The row and column indexes of the resulting DataFrame will be the union of the two.

**Parameters**

other [DataFrame] Provided DataFrame to use to fill null values.

**Returns**

combined [DataFrame]

See also:

*DataFrame.combine* Perform series-wise operation on two DataFrames using a given function.

**Examples**

```python
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})  # doctest: +SKIP
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})  # doctest: +SKIP

>>> df1.combine_first(df2)  # doctest: +SKIP
        A  B
      0 1.0 3.0
      1 0.0 4.0

Null values still persist if the location of that null value does not exist in other

```python
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})  # doctest: +SKIP

```python
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])  # doctest: +SKIP

```python
>>> df1.combine_first(df2)  # doctest: +SKIP

        A  B  C
      0  NaN 4.0  NaN
      1  0.0 3.0  1.0
      2  NaN 3.0  1.0

**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**

scheduler [string, optional] Which scheduler to use like “threads”, “synchronous” or “processes”. 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 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.

```python
corr (method='pearson', min_periods=None, split_every=False)
```
Compute pairwise correlation of columns, excluding NA/null values.

This docstring was copied from pandas.core.frame.DataFrame.corr.
Some inconsistencies with the Dask version may exist.

**Parameters**

- `method` : [{'pearson', 'kendall', 'spearman'} or callable]
  - pearson : standard correlation coefficient
  - kendall : Kendall Tau correlation coefficient
  - spearman : Spearman rank correlation
  - callable: callable with input two 1d ndarrays and returning a float .. version-added:: 0.24.0

- `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]

**See also:**

DataFrame.corrwith, Series.corr

**Examples**

```python
>>> histogram_intersection = lambda a, b: np.minimum(a, b)  # doctest: +SKIP
... .sum().round(decimals=1)
>>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)], columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)  # doctest: +SKIP
dogs cats
dogs 1.0 0.3
cats 0.3 1.0
```

```python
count (axis=None, split_every=False)
```
Count non-NA cells for each column or row.

This docstring was copied from pandas.core.frame.DataFrame.count.
Some inconsistencies with the Dask version may exist.

The values `None`, `NaN`, `NaT`, and optionally `numpy.inf` (depending on `pandas.options.mode.use_inf_as_na`) are considered NA.

**Parameters**

- `axis` : [0 or ‘index’, 1 or ‘columns’], default 0] If 0 or ‘index’ counts are generated for each column. If 1 or ‘columns’ counts are generated for each row.
level [int or str, optional (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame. A str specifies the level name.

numeric_only [boolean, default False (Not supported in Dask)] Include only float, int or boolean data.

Returns

Series or DataFrame For each column/row the number of non-NA/null entries. If level is specified returns a DataFrame.

See also:

Series.count Number of non-NA elements in a Series.

DataFrame.shape Number of DataFrame rows and columns (including NA elements).

DataFrame.isna Boolean same-sized DataFrame showing places of NA elements.

Examples

Constructing DataFrame from a dictionary:

```python
```

Notice the uncounted NA values:

```python
>>> df.count()  # doctest: +SKIP
Person  5
Age    4
Single  5
dtype: int64
```

Counts for each row:

```python
>>> df.count(axis='columns')  # doctest: +SKIP
0   3
1   2
2   3
3   3
4   3
dtype: int64
```

Counts for one level of a MultiIndex:
cov (min_periods=\text{None}, \text{split}_{\text{every}}=\text{False})
Compute pairwise covariance of columns, excluding NA/null values.

This docstring was copied from pandas.core.frame.DataFrame.cov.

Some inconsistencies with the Dask version may exist.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as NaN.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

Parameters

min\_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

Returns

Dataframe The covariance matrix of the series of the DataFrame.

See also:

pandas.Series.cov Compute covariance with another Series.
pandas.core.window.EWM.cov Exponential weighted sample covariance.
pandas.core.window.Expanding.cov Expanding sample covariance.
pandas.core.window.Rolling.cov Rolling sample covariance.

Notes

Returns the covariance matrix of the DataFrame’s time series. The covariance is normalized by N\(-1\).

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.
Examples

```python
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
                     columns=['dogs', 'cats'])
>>> df.cov()  # doctest: +SKIP
dogs   cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```python
>>> np.random.seed(42)  # doctest: +SKIP
>>> df = pd.DataFrame(np.random.randn(1000, 5),
                     columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()  # doctest: +SKIP
    a     b     c     d     e
a  0.998438 -0.020161 0.059277 -0.008943 0.014144
b -0.020161  1.059352 -0.008543 -0.024738 0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```python
>>> np.random.seed(42)  # doctest: +SKIP
>>> df = pd.DataFrame(np.random.randn(20, 3),
                     columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan  # doctest: +SKIP
>>> df.loc[df.index[5:10], 'b'] = np.nan  # doctest: +SKIP
>>> df.cov(min_periods=12)  # doctest: +SKIP
    a     b     c
a  0.316741 NaN -0.150812
b  NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

cummax

Return cumulative maximum over a DataFrame or Series axis.

This docstring was copied from pandas.core.frame.DataFrame.cummax.

Some inconsistencies with the Dask version may exist.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

- **axis** ([0 or ‘index’, 1 or ‘columns’], default 0) The index or the name of the axis. 0 is equivalent to None or ‘index’.
- **skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- *args, **kwargs : Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

- **cummax** [Series or DataFrame]

See also:
**core.window.Expanding.max** Similar functionality but ignores NaN values.

**DataFrame.max** Return the maximum over DataFrame axis.

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

### Examples

#### Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0   2.0
1   NaN
2   5.0
3  -1.0
4   0.0
dtype: float64
```

By default, NA values are ignored.

```python
>>> s.cummax()  # doctest: +SKIP
0  2.0
1  NaN
2  5.0
3  5.0
4  5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```python
>>> s.cummax(skipna=False)  # doctest: +SKIP
0  2.0
1  NaN
2  NaN
3  NaN
4  NaN
dtype: float64
```

#### DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0],  # doctest: +SKIP
...                     [3.0, np.nan],
...                     [1.0, 0.0]],
...                     columns=list('AB'))
>>> df  # doctest: +SKIP
  A  B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`. 

428 Chapter 4. Index
>>> df.cummax()  # doctest: +SKIP
   A  B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0

To iterate over columns and find the maximum in each row, use axis=1

>>> df.cummax(axis=1)  # doctest: +SKIP
   A  B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0

cummin (axis=None, skipna=True, out=None)

Return cumulative minimum over a DataFrame or Series axis.

This docstring was copied from pandas.core.frame.DataFrame.cummin.

Some inconsistencies with the Dask version may exist.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

*args, **kwargs : Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummin [Series or DataFrame]

See also:

core.window.Expanding.min Similar functionality but ignores NaN values.

DataFrame.min Return the minimum over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0    2.0
1  NaN
2    5.0

(continues on next page)
By default, NA values are ignored.

```python
>>> s.cummin()   # doctest: +SKIP
    0  2.0
    1 NaN
    2  2.0
    3 -1.0
    4 -1.0
dtype: float64
```

To include NA values in the operation, use skipna=False

```python
>>> s.cummin(skipna=False)   # doctest: +SKIP
    0  2.0
    1 NaN
    2 NaN
    3 NaN
    4 NaN
dtype: float64
```

DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0], ...
... [3.0, np.nan], ...
... [1.0, 0.0]], ...
... columns=list('AB'))
```

```python
>>> df   # doctest: +SKIP
     A  B
 0  2.0  1.0
 1  3.0  NaN
 2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to axis=None or axis='index'.

```python
>>> df.cummin()   # doctest: +SKIP
     A  B
 0  2.0  1.0
 1  2.0  NaN
 2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use axis=1

```python
>>> df.cummin(axis=1)   # doctest: +SKIP
     A  B
 0  2.0  1.0
 1  3.0  NaN
 2  1.0  0.0
```

cumprod(axis=None, skipna=True, dtype=None, out=None)

Return cumulative product over a DataFrame or Series axis.

This docstring was copied from pandas.core.frame.DataFrame.cumprod.
Some inconsistencies with the Dask version may exist.

Returns a DataFrame or Series of the same size containing the cumulative product.

**Parameters**

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.
- **skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **args**, **kwargs**: Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **cumprod** [Series or DataFrame]

**See also:**

- [core.window.Expanding.prod](#) Similar functionality but ignores NaN values.
- [DataFrame.prod](#) Return the product over DataFrame axis.
- [DataFrame.cummax](#) Return cumulative maximum over DataFrame axis.
- [DataFrame.cummin](#) Return cumulative minimum over DataFrame axis.
- [DataFrame.cumsum](#) Return cumulative sum over DataFrame axis.
- [DataFrame.cumprod](#) Return cumulative product over DataFrame axis.

**Examples**

**Series**

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0    2.0
1   NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```python
>>> s.cumprod()  # doctest: +SKIP
0    2.0
1    NaN
2    10.0
3  -10.0
4     0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```python
>>> s.cumprod(skipna=False)  # doctest: +SKIP
0    2.0
1   NaN
```

(continues on next page)
DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0],
                     [3.0, np.nan],
                     [1.0, 0.0]],
                   columns=list('AB'))
>>> df
   A   B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```python
>>> df.cumprod()
   A   B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```python
>>> df.cumprod(axis=1)
   A   B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

`cumsum` (axis=None, skipna=True, dtype=None, out=None)

Return cumulative sum over a DataFrame or Series axis.

This docstring was copied from pandas.core.frame.DataFrame.cumsum.

Some inconsistencies with the Dask version may exist.

Returns a DataFrame or Series of the same size containing the cumulative sum.

**Parameters**

- `axis` [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.
- `skipna` [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- `*args`, `**kwargs` Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- `cumsum` [Series or DataFrame]

See also:
**core.window.Expanding.sum**  Similar functionality but ignores NaN values.

**DataFrame.sum**  Return the sum over DataFrame axis.

**DataFrame.cummax**  Return cumulative maximum over DataFrame axis.

**DataFrame.cummin**  Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum**  Return cumulative sum over DataFrame axis.

**DataFrame.cumprod**  Return cumulative product over DataFrame axis.

### Examples

#### Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0  2.0
1  NaN
2  5.0
3 -1.0
4  0.0
dtype: float64
```

By default, NA values are ignored.

```python
>>> s.cumsum()  # doctest: +SKIP
0  2.0
1  NaN
2  7.0
3  6.0
4  6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```python
>>> s.cumsum(skipna=False)  # doctest: +SKIP
0  2.0
1  NaN
2  NaN
3  NaN
4  NaN
dtype: float64
```

#### DataFrame

```python
>>> df = pd.DataFrame([[2.0, 1.0], ... [3.0, np.nan], ... [1.0, 0.0]], columns=list('AB'))
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.
To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)  # doctest: +SKIP
A  B
0  2.0  3.0
1  3.0 NaN
2  1.0  1.0
```
See also:

**DataFrame.count**  Count number of non-NA/null observations.

**DataFrame.max**  Maximum of the values in the object.

**DataFrame.min**  Minimum of the values in the object.

**DataFrame.mean**  Mean of the values.

**DataFrame.std**  Standard deviation of the observations.

**DataFrame.select_dtypes**  Subset of a DataFrame including/excluding columns based on their dtype.

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.

```python
>>> s = pd.Series([1, 2, 3])  # doctest: +SKIP
>>> s.describe()  # doctest: +SKIP
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical Series.

```python
>>> s = pd.Series(['a', 'a', 'b', 'c'])  # doctest: +SKIP
>>> s.describe()  # doctest: +SKIP
count    4
unique   3
```

(continues on next page)
Describing a timestamp Series.

```
>>> s = pd.Series([  
...     np.datetime64("2000-01-01"),  
...     np.datetime64("2010-01-01"),  
...     np.datetime64("2010-01-01")  
... ])  
>>> s.describe()  
... # doctest: +SKIP  
... 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({'categorical': pd.Categorical(['d', 'e', 'f']),  
...     'numeric': [1, 2, 3],  
...     'object': ['a', 'b', 'c']  
... })  
>>> df.describe()  
... # doctest: +SKIP  
... 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')  
... # doctest: +SKIP  
... 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.
Including only numeric columns in a DataFrame description.

```python
>>> df.describe(include=[np.number])  # doctest: +SKIP
    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
Name: numeric, dtype: float64
```

Including only string columns in a DataFrame description.

```python
>>> df.describe(include=[np.object])  # doctest: +SKIP
    object
count  3
unique 3
top   c
freq  1
```

Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])  # doctest: +SKIP
    categorical
count  3
unique 3
top   f
freq  1
```

Excluding numeric columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.number])  # doctest: +SKIP
    categorical object
count  3  3
unique 3  3
top   f  c
freq  1  1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.object])  # doctest: +SKIP
    categorical numeric
count  3  3.0
unique 3  NaN
(continues on next page)
**diff** *(periods=1, axis=0)*

First discrete difference of element.

This docstring was copied from pandas.core.frame.DataFrame.diff.

Some inconsistencies with the Dask version may exist.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

**Parameters**

- **periods** [int, default 1] Periods to shift for calculating difference, accepts negative values.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

**Returns**

- **diffed** [DataFrame]

**See also:**

- **Series.diff** First discrete difference for a Series.
- **DataFrame.pct_change** Percent change over given number of periods.
- **DataFrame.shift** Shift index by desired number of periods with an optional time freq.

**Examples**

Difference with previous row

```python
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],  # doctest: +SKIP
...                   'b': [1, 1, 2, 3, 5, 8],
...                   'c': [1, 4, 9, 16, 25, 36]})
```

```python
a  b  c
0 1 1 1
1 2 1 4
2 3 2 9
3 4 3 16
4 5 5 25
5 6 8 36
```
>>> df.diff()  # doctest: +SKIP
     a    b    c
 0   NaN  NaN  NaN
 1   1.0  0.0  3.0
 2   1.0  1.0  5.0
 3   1.0  1.0  7.0
 4   1.0  2.0  9.0
 5   1.0  3.0 11.0

Difference with previous column

>>> df.diff(axis=1)  # doctest: +SKIP
     a    b    c
 0   NaN  0.0  0.0
 1   NaN -1.0  3.0
 2   NaN -1.0  7.0
 3   NaN -1.0 13.0
 4   NaN  0.0 20.0
 5   NaN  2.0 28.0

Difference with 3rd previous row

>>> df.diff(periods=3)  # doctest: +SKIP
     a    b    c
 0   NaN  NaN  NaN
 1   NaN  NaN  NaN
 2   NaN  NaN  NaN
 3   3.0  2.0 15.0
 4   3.0  4.0 21.0
 5   3.0  6.0 27.0

Difference with following row

>>> df.diff(periods=-1)  # doctest: +SKIP
     a    b    c
 0  -1.0  0.0 -3.0
 1  -1.0 -1.0 -5.0
 2  -1.0 -1.0 -7.0
 3  -1.0 -2.0 -9.0
 4  -1.0 -3.0 -11.0
 5   NaN  NaN  NaN

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. With reverse version, rtruediv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
axis [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
level [int or label] Broadcast across a level, matching Index values on the passed Multi-Index level.
**fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

DataFrame Result of the arithmetic operation.

**See also:**

- **DataFrame.add** Add DataFrames.
- **DataFrame.sub** Subtract DataFrames.
- **DataFrame.mul** Multiply DataFrames.
- **DataFrame.div** Divide DataFrames (float division).
- **DataFrame.truediv** Divide DataFrames (float division).
- **DataFrame.floordiv** Divide DataFrames (integer division).
- **DataFrame.mod** Calculate modulo (remainder after division).
- **DataFrame.pow** Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
                    'degrees': [360, 180, 360]},
                   index=['circle', 'triangle', 'rectangle'])
```

```plaintext
    angles  degrees
circle    0      360
triangle   3      180
rectangle  4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
    angles  degrees
circle    1      361
triangle   4      181
rectangle  5      361
```

```python
>>> df.add(1)  # doctest: +SKIP
    angles  degrees
circle    1      361
triangle   4      181
rectangle  5      361
```

Divide by constant with reverse version.
```python
>>> df.div(10)  # doctest: +SKIP
   angles  degrees
   circle   0.0    36.0
      triangle  0.3    18.0
             rectangle  0.4    36.0

>>> df.rdiv(10)  # doctest: +SKIP
   angles    degrees
   circle    inf    0.027778
           triangle  3.333333   0.055556
            rectangle  2.500000   0.027778

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
   angles  degrees
   circle   -1    358
      triangle    2    178
             rectangle    3    358

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
   angles  degrees
   circle   -1    358
      triangle    2    178
             rectangle    3    358

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),  # doctest: +SKIP...
   axis='index')
   angles  degrees
   circle   -1    359
      triangle    2    179
             rectangle    3    359

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4]},  # doctest: +SKIP...
   index=['circle', 'triangle', 'rectangle'])

>>> df * other  # doctest: +SKIP
   angles  degrees
   circle    0     NaN
      triangle    9     NaN
             rectangle   16     NaN

>>> df.mul(other, fill_value=0)  # doctest: +SKIP
   angles  degrees
   circle    0     0.0
      triangle    9     0.0
             rectangle   16     0.0
```

4.9. DataFrame 441
Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
                               'degrees': [360, 180, 360, 360, 540, 720],
                               index=[['A', 'A', 'A', 'B', 'B', 'B'],
                                      ['circle', 'triangle', 'rectangle',
                                       'square', 'pentagon', 'hexagon']])
```

```python
>>> df_multindex
angles  degrees
A  circle     0  360
    triangle   3  180
    rectangle  4  360
B  square    4  360
    pentagon  5  540
    hexagon  6  720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)
```

```python
angles  degrees
A  circle  NaN  1.0
    triangle  1.0  1.0
    rectangle  1.0  1.0
B  square  0.0  0.0
    pentagon  0.0  0.0
    hexagon  0.0  0.0
```

**drop** *(labels, axis=0, errors=’raise’)*

Drop specified labels from rows or columns.

This docstring was copied from pandas.core.frame.DataFrame.drop.

Some inconsistencies with the Dask version may exist.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly
index or column names. When using a multi-index, labels on different levels can be removed by specifying
the level.

**Parameters**

- **labels** [single label or list-like] Index or column labels to drop.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] Whether to drop labels from the index (0
  or ‘index’) or columns (1 or ‘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, optional (Not supported in Dask)] For MultiIndex, level from
  which the labels will be removed.
- **inplace** [bool, default False (Not supported in Dask)] If True, do operation inplace and
  return None.
- **errors** [{‘ignore’, ‘raise’}, default ‘raise’] If ‘ignore’, suppress error and only existing
  labels are dropped.

**Returns**

- **dropped** [pandas.DataFrame]

**Raises**
**KeyError** If none of the labels are found in the selected axis

See also:

- `DataFrame.loc` Label-location based indexer for selection by label.
- `DataFrame.dropna` Return DataFrame with labels on given axis omitted where (all or any) data are missing.
- `DataFrame.drop_duplicates` Return DataFrame with duplicate rows removed, optionally only considering certain columns.
- `Series.drop` Return Series with specified index labels removed.

## Examples

```python
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),  # doctest: +SKIP
...     columns=['A', 'B', 'C', 'D'])
```

Drop columns

```python
>>> df.drop(['B', 'C'], axis=1)  # doctest: +SKIP
    A  D
0  0  3
1  4  7
2  8 11
```

Drop a row by index

```python
>>> df.drop([0, 1])  # doctest: +SKIP
    A  B  C  D
2  8  9 10 11
```

Drop columns and/or rows of MultiIndex DataFrame

```python
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],  # doctest: +SKIP
...      ['speed', 'weight', 'length']],
...     ...     codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...      ...      [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],  # doctest: +SKIP
...     ...     data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...      ...      [250, 150], [1.5, 0.8], [320, 250],
...      ...      [1, 0.8], [0.3,0.2]])
>>> df  # doctest: +SKIP
    big  small
lama  speed  45.0  30.0
```

(continues on next page)
```python
>>> df.drop(index='cow', columns='small')  # doctest: +SKIP
big
lama speed 45.0
weight 200.0
length 1.5
falcon speed 320.0
weight 1.0
length 0.3
```

```python
>>> df.drop(index='length', level=1)  # doctest: +SKIP
big small
lama speed 45.0 30.0
weight 200.0 100.0
cow speed 30.0 20.0
weight 250.0 150.0
falcon speed 320.0 250.0
weight 1.0 0.8
```
Parameters

axis [(0 or 'index', 1 or 'columns'), default 0 (Not supported in Dask)] Determine if rows or columns which contain missing values are removed.
  • 0, or 'index': Drop rows which contain missing values.
  • 1, or 'columns': Drop columns which contain missing value.

Deprecated since version 0.23.0: Pass tuple or list to drop on multiple axes. Only a single axis is allowed.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.
  • 'any': If any NA values are present, drop that row or column.
  • 'all': If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False (Not supported in Dask)] If True, do operation inplace and return None.

Returns

DataFrame DataFrame with NA entries dropped from it.

See also:

DataFrame.isna Indicate missing values.
DataFrame.notna Indicate existing (non-missing) values.
DataFrame.fillna Replace missing values.
Series.dropna Drop missing values.
Index.dropna Drop missing indices.

Examples

```python
>>> df = pd.DataFrame({'name': ['Alfred', 'Batman', 'Catwoman'],
  ...                     'toy': [np.nan, 'Batmobile', 'Bullwhip'],
  ...                     'born': [pd.NaT, pd.Timestamp('1940-04-25'),
  ...                     pd.NaT])
>>> df  # doctest: +SKIP
   name      toy      born
0  Alfred    NaN  NaT
1  Batman  Batmobile  1940-04-25
2  Catwoman  Bullwhip  NaT
```

Drop the rows where at least one element is missing.

```python
>>> df.dropna()  # doctest: +SKIP
   name      toy      born
0  Alfred    NaN  NaT
1  Batman  Batmobile  1940-04-25
```

Drop the columns where at least one element is missing.

```python
>>> df.dropna()  # doctest: +SKIP
   name      toy      born
0  Alfred    NaN  NaT
1  Batman  Batmobile  1940-04-25
```
Drop the rows where all elements are missing.

```python
>>> df.dropna(axis='columns')  # doctest: +SKIP
name
0  Alfred
1  Batman
2  Catwoman
```

Keep only the rows with at least 2 non-NA values.

```python
>>> df.dropna(how='all')  # doctest: +SKIP
name  toy  born
0  Alfred  NaN  NaT
1  Batman  Batmobile  1940-04-25
2  Catwoman  Bullwhip  NaT
```

Define in which columns to look for missing values.

```python
>>> df.dropna(subset=['name', 'born'])  # doctest: +SKIP
name  toy  born
1  Batman  Batmobile  1940-04-25
2  Catwoman  Bullwhip  NaT
```

Keep the DataFrame with valid entries in the same variable.

```python
>>> df.dropna(inplace=True)  # doctest: +SKIP
```

```
name  toy  born
1  Batman  Batmobile  1940-04-25
```

dtypes

Return data types

eq (other, axis='columns', level=None)

Equal to of dataframe and other, element-wise (binary operator eq).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:
DataFrame.eq  Compare DataFrames for equality elementwise.

DataFrame.ne  Compare DataFrames for inequality elementwise.

DataFrame.le  Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt  Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge  Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt  Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

Comparison with a scalar, using either the operator or method:

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100], 'revenue': [100, 250, 300]}, index=['A', 'B', 'C'])
>>> df  # doctest: +SKIP
   cost  revenue
  A   250     100
  B   150     250
  C   100     300

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100  # doctest: +SKIP
   cost  revenue
  A   False    True
  B   False   False
  C    True    False

>>> df.eq(100)  # doctest: +SKIP
   cost  revenue
  A   False    True
  B   False   False
  C    True    False
```

When other is a Series, the columns of a DataFrame are aligned with the index of other and broadcast:

```python
>>> df != pd.Series([100, 250], index=['A', 'B', 'C'])  # doctest: +SKIP
   cost  revenue
  A     True    False
  B     True    False
  C    False     True

Use the method to control the broadcast axis:

```python
>>> df.ne(pd.Series([100, 300], index=['A', 'D']), axis='index')  # doctest: +SKIP
   cost  revenue
  A     True    False
  B     True     True
```

(continues on next page)
When comparing to an arbitrary sequence, the number of columns must match the number elements in `other`:

```python
>>> df == [250, 100]  # doctest: +SKIP
    cost    revenue
  A True     True
  B False    False
  C False    False

Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')  # doctest: +SKIP
    cost    revenue
  A True     False
  B False     True
  C True     False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150],
                        'index': ['A', 'B', 'C', 'D']})
```

```python
>>> df.gt(other)  # doctest: +SKIP
    cost    revenue
  A False    False
  B False    False
  C False     True
  D False    False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
                                'revenue': [100, 250, 300, 200, 175, 225],
                                'index': ['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2']})
```

```python
>>> df_multindex.le(df_multindex, level=1)  # doctest: +SKIP
    cost    revenue
  Q1 A  250    100
    B  150    250
    C  100    300
  Q2 A  150    200
    B  300    175
    C  220    225
```

(continues on next page)
**eval** (*expr, inplace=None, **kwargs*)

Evaluate a string describing operations on DataFrame columns.

This docstring was copied from pandas.core.frame.DataFrame.eval.

Some inconsistencies with the Dask version may exist.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

**Parameters**

- **expr** (*str*) The expression string to evaluate.
- **inplace** (*bool, default False*) If the expression contains an assignment, whether to perform the operation inplace 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**

- *ndarray, scalar, or pandas object* The result of the evaluation.

**See also:**

- **DataFrame.query** Evaluates a boolean expression to query the columns of a frame.
- **DataFrame.assign** Can evaluate an expression or function to create new values for a column.
- **pandas.eval** Evaluate a Python expression as a string using various backends.

**Notes**

For more details see the API documentation for *eval()*.

For detailed examples see *enhancing performance with eval*.

**Examples**

```python
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})  # doctest:+SKIP
>>> df  # doctest: +SKIP
        A  B
   0 1 10
   1 2 8
   2 3 6
   3 4 4
   4 5 2
```

(continues on next page)
df.eval('A + B') # doctest: +SKIP
0   11
1   10
2    9
3    8
4    7
dtype: int64

Assignment is allowed though by default the original DataFrame is not modified.

df.eval('C = A + B') # doctest: +SKIP
A  B  C
0  1  10 11
1  2   8 10
2  3   6  9
3  4   4  8
4  5   2  7

Use inplace=True to modify the original DataFrame.

df.eval('C = A + B', inplace=True) # doctest: +SKIP
df # doctest: +SKIP
A  B  C
0  1  10 11
1  2   8 10
2  3   6  9
3  4   4  8
4  5   2  7

ffill (axis=None, limit=None)
Synonym for DataFrame.fillna() with method='ffill'.

fillna (value=None, method=None, limit=None, axis=None)
Fill NA/NaN values using the specified method.

This docstring was copied from pandas.core.frame.DataFrame.fillna.

Some inconsistencies with the Dask version may exist.

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.

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 (Not supported in Dask)] 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 (Not supported in Dask)] 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:

interpolate Fill NaN values using interpolation.

reindex, asfreq

Examples

```python
gf = 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'))
gf
```

Replace all NaN elements with 0s.

```python
gf.fillna(0)
```

We can also propagate non-null values forward or backward.

```python
gf.fillna(method='ffill')
```

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}  # doctest: +SKIP
>>> df.fillna(value=values)  # doctest: +SKIP
       A   B   C   D
0   0.0  2.0  2.0  0.0
1   3.0  4.0  2.0  1.0
2   0.0  1.0  2.0  5.0
3   0.0  3.0  2.0  4.0

Only replace the first NaN element.
```
```
>>> df.fillna(value=values, limit=1)  # doctest: +SKIP
       A   B   C   D
0   0.0  2.0  2.0  0.0
1   3.0  4.0  NaN  1.0
2   NaN  1.0  NaN  5.0
3   NaN  3.0  NaN  4.0
```

**first** *(offset)*

Convenience method for subsetting initial periods of time series data based on a date offset.

This docstring was copied from pandas.core.frame.DataFrame.first.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **offset** [string, DateOffset, dateutil.relativedelta]

**Returns**

- **subset** [same type as caller]

**Raises**

- **TypeError** If the index is not a DatetimeIndex

**See also:**

- **last** Select final periods of time series based on a date offset.
- **at_time** Select values at a particular time of the day.
- **between_time** Select values between particular times of the day.

**Examples**

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')  # doctest: +SKIP
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)  # doctest: +SKIP
>>> ts  # doctest: +SKIP
     A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:
Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

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. With reverse version, rfloordiv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %. **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [(0 or ‘index’, 1 or ‘columns’)] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.
Examples

```python
df = pd.DataFrame(
    {'angles': [0, 3, 4],  # doctest: +SKIP
    'degrees': [360, 180, 360]},
    index=['circle', 'triangle', 'rectangle'])
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
      angles  degrees
    circle    1     361
    triangle   4     181
    rectangle   5     361
```

```python
>>> df.add(1)  # doctest: +SKIP
      angles  degrees
    circle    1     361
    triangle   4     181
    rectangle   5     361
```

Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
      angles  degrees
    circle 0.0    36.0
    triangle 0.3    18.0
    rectangle 0.4    36.0
```

```python
>>> df.rdiv(10)  # doctest: +SKIP
      angles  degrees
    circle  inf     0.027778
    triangle 3.333333     0.055556
    rectangle 2.500000     0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
      angles  degrees
    circle   -1     358
    triangle    2     178
    rectangle    3     358
```

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
      angles  degrees
    circle   -1     358
    triangle    2     178
    rectangle    3     358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),  # doctest: +SKIP
    axis='index')
```

(continues on next page)
Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4]},
# doctest: +SKIP
index=['circle', 'triangle', 'rectangle'])
```

```python
df * other  # doctest: +SKIP
angles
circle 0  NaN
triangle 9  NaN
rectangle 16  NaN
```

```python
df.mul(other, fill_value=0)  # doctest: +SKIP
angles
circle 0  0.0
triangle 9  0.0
rectangle 16  0.0
```

Divide by a MultiIndex by level.

```python
df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
# doctest: +SKIP
... 'degrees': [360, 180, 360, 360, 540, 720]},
... index=[['A', 'A', 'A', 'B', 'B', 'B'],
... ['circle', 'triangle', 'rectangle',
... 'square', 'pentagon', 'hexagon']])
```

```python
df.div(df_multindex, level=1, fill_value=0)  # doctest: +SKIP
angles
circle NaN  1.0
triangle 1.0  1.0
rectangle 1.0  1.0
B square 0.0  0.0
pentagon 0.0  0.0
hexagon 0.0  0.0
```

ge (other, axis='columns', level=None)
Greater than or equal to of dataframe and other, element-wise (binary operator ge).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.
Equivalent to $==$, $!=$, $<=$, $<$, $>=$, $>$ with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or ‘index’, 1 or ‘columns’}, default ‘columns’] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).
- **level** [int or label] Broadcast across a level, matching Index values on the passed Multi-Index level.

**Returns**

- **DataFrame of bool** Result of the comparison.

**See also:**

- `DataFrame.eq` Compare DataFrames for equality elementwise.
- `DataFrame.ne` Compare DataFrames for inequality elementwise.
- `DataFrame.le` Compare DataFrames for less than inequality or equality elementwise.
- `DataFrame.lt` Compare DataFrames for strictly less than inequality elementwise.
- `DataFrame.ge` Compare DataFrames for greater than inequality or equality elementwise.
- `DataFrame.gt` Compare DataFrames for strictly greater than inequality elementwise.

**Notes**

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* $\neq NaN$).

**Examples**

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100],
                    'revenue': [100, 250, 300]},
                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250     100
B   150     250
C   100     300
```

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100  # doctest: +SKIP
   cost  revenue
A  False   True
B  False  False
C   True  False
```

```python
>>> df.eq(100)  # doctest: +SKIP
   cost  revenue
A  False   True
```
When \( \text{other} \) is a \texttt{Series}, the columns of a DataFrame are aligned with the index of \( \text{other} \) and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
# doctest: +SKIP
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>True</td>
</tr>
<tr>
<td>C</td>
<td>False</td>
</tr>
</tbody>
</table>

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
# doctest: +SKIP
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>True</td>
</tr>
<tr>
<td>C</td>
<td>True</td>
</tr>
<tr>
<td>D</td>
<td>True</td>
</tr>
</tbody>
</table>
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in \( \text{other} \):

```
>>> df == [250, 100]
# doctest: +SKIP
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>False</td>
</tr>
</tbody>
</table>

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
# doctest: +SKIP
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>False</td>
</tr>
<tr>
<td>B</td>
<td>True</td>
</tr>
<tr>
<td>C</td>
<td>True</td>
</tr>
</tbody>
</table>
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                       index=["A", "B", "C", "D"])
```

```
>>> df.gt(other)
# doctest: +SKIP
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td>revenue</td>
</tr>
<tr>
<td>A</td>
<td>False</td>
</tr>
<tr>
<td>B</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>True</td>
</tr>
<tr>
<td>D</td>
<td>False</td>
</tr>
</tbody>
</table>
```

Compare to a MultiIndex by level.
df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
    'revenue': [100, 250, 300, 200, 175, 225]},
    index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
    ['A', 'B', 'C', 'A', 'B', 'C']])

df_multindex

cost revenue
Q1 A 250 100
    B 150 250
    C 100 300
Q2 A 150 200
    B 300 175
    C 220 225

df.le(df_multindex, level=1)

cost revenue
Q1 A True True
    B True True
    C True True
Q2 A False True
    B True False
    C True False

def.get_dtype_counts()
Return counts of unique dtypes in this object.

This docstring was copied from pandas.core.frame.DataFrame.get_dtypes.

Some inconsistencies with the Dask version may exist.

Returns
   dtype [Series] Series with the count of columns with each dtype.

See also:
   dtypes Return the dtypes in this object.

Examples

>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]  # doctest: +SKIP
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])  # doctest: +SKIP
>>> df # doctest: +SKIP
   str  int  float
  0   a    1   1.0
  1   b    2   2.0
  2   c    3   3.0

>>> df.get_dtype_counts()  # doctest: +SKIP
float64 1
int64   1
object  1
dtype: int64

def.get_ftype_counts()
Return counts of unique ftypes in this object.

This docstring was copied from pandas.core.frame.DataFrame.get_ftypes.


Some inconsistencies with the Dask version may exist.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

**Returns**

- `dtype` [Series] Series with the count of columns with each type and sparsity (dense/sparse)

**See also:**

- `ftypes` Return ftypes (indication of sparse/dense and dtype) in this object.

**Examples**

```python
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]  # doctest: +SKIP
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])  # doctest: +SKIP
>>> df
str  int  float
0   a   1    1.0
1   b   2    2.0
2   c   3    3.0

>>> df.get_ftype_counts()  # doctest: +SKIP
float64:dense 1
int64:dense   1
object:dense  1
dtype: int64
```

**get_partition** *(n)*  
Get a dask DataFrame/Series representing the *nth* partition.

**groupby** *(by=None, **kwargs)*  
Group DataFrame or Series using a mapper or by a Series of columns.

This docstring was copied from pandas.core.frame.DataFrame.groupby.

Some inconsistencies with the Dask version may exist.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

**Parameters**

- `by` [mapping, function, label, or list of labels] 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 label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

- `axis` [[0 or ‘index’, 1 or ‘columns’], default 0 (Not supported in Dask)] Split along rows (0) or columns (1).

- `level` [int, level name, or sequence of such, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), group by a particular level or levels.
as_index [bool, default True (Not supported in Dask)] 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 [bool, default True (Not supported in Dask)] 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 [bool, default True (Not supported in Dask)] When calling apply, add group keys to index to identify pieces.

squeeze [bool, default False (Not supported in Dask)] Reduce the dimensionality of the return type if possible, otherwise return a consistent type.

observed [bool, default False (Not supported in Dask)] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

**kwargs Optional, only accepts keyword argument ‘mutated’ and is passed to groupby.

Returns

DataFrameGroupBy or SeriesGroupBy Depends on the calling object and returns groupby object that contains information about the groups.

See also:

resample Convenience method for frequency conversion and resampling of time series.

Notes

See the user guide for more.

Examples

```python
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon', # doctest: +SKIP
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df # doctest: +SKIP
     Animal  Max Speed
0     Falcon   380.0
1     Falcon   370.0
2    Parrot    24.0
3    Parrot    26.0
>>> df.groupby(['Animal']).mean() # doctest: +SKIP
     Max Speed
Animal
Falcon   375.0
Parrot   25.0
```

Hierarchical Indexes

We can groupby different levels of a hierarchical index using the level parameter:
```python
>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
           ['Capitve', 'Wild', 'Capitve', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
                   index=index)
```

```
   Animal  Type  Max Speed
---  ----  -------
  Falcon  Capitve    390.0
         Wild     350.0
 Parrot  Capitve     30.0
         Wild     20.0
```

```python
>>> df.groupby(level=0).mean()  # doctest: +SKIP
```

```
   Max Speed
Animal
  Falcon    370.0
  Parrot    25.0
```

```python
>>> df.groupby(level=1).mean()  # doctest: +SKIP
```

```
   Max Speed
Type
  Capitve     210.0
   Wild      185.0
```

```python
gt (other, axis='columns', level=None)
Greater than of dataframe and other, element-wise (binary operator gt).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.
Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or 'index', 1 or 'columns'}, default 'columns'] Whether to compare by the index (0 or 'index') or columns (1 or 'columns').
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

- **DataFrame of bool** Result of the comparison.

See also:

- **DataFrame.eq** Compare DataFrames for equality elementwise.
- **DataFrame.ne** Compare DataFrames for inequality elementwise.
- **DataFrame.le** Compare DataFrames for less than inequality or equality elementwise.
- **DataFrame.lt** Compare DataFrames for strictly less than inequality elementwise.
- **DataFrame.ge** Compare DataFrames for greater than inequality or equality elementwise.
- **DataFrame.gt** Compare DataFrames for strictly greater than inequality elementwise.
Notes

Mismatched indices will be unioned together. \textit{NaN} values are considered different (i.e. \textit{NaN} \neq \textit{NaN}).

Examples

\begin{verbatim}
>>> df = pd.DataFrame({'cost': [250, 150, 100],  # doctest: +SKIP
...    'revenue': [100, 250, 300]},
...    index=['A', 'B', 'C'])

>>> df  # doctest: +SKIP
    cost  revenue
  A     250     100
  B     150     250
  C     100     300

Comparison with a scalar, using either the operator or method:

\begin{verbatim}
>>> df == 100  # doctest: +SKIP
    cost  revenue
  A   False    True
  B   False   False
  C    True   False

>>> df.eq(100)  # doctest: +SKIP
    cost  revenue
  A   False    True
  B   False   False
  C    True   False
\end{verbatim}

When \textit{other} is a \textit{Series}, the columns of a DataFrame are aligned with the index of \textit{other} and broadcast:

\begin{verbatim}
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])  # doctest: +SKIP
    cost  revenue
  A    True    True
  B    True   False
  C   False    True

Use the method to control the broadcast axis:

\begin{verbatim}
>>> df.ne(pd.Series([100, 300], index=['A', 'D']), axis='index')  # doctest: +SKIP
    cost  revenue
  A    True   False
  B    True    True
  C    True    True
  D    True    True
\end{verbatim}

When comparing to an arbitrary sequence, the number of columns must match the number elements in \textit{other}:

\begin{verbatim}
>>> df == [250, 100]  # doctest: +SKIP
    cost  revenue
  A    True    True
  B   False   False
  C   False   False
\end{verbatim}
Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')  # doctest: +SKIP
    cost  revenue
   A   True  False
   B   False  True
   C   True  False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                        index=['A', 'B', 'C', 'D'])
>>> other
    revenue
   A    300
   B    250
   C    100
   D    150
```

```python
>>> df.gt(other)  # doctest: +SKIP
    cost  revenue
   A   False  False
   B   False  False
   C   False  True
   D   False  False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                             'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                   ['A', 'B', 'C', 'A', 'B', 'C']])
```

```python
>>> df_multindex
   cost  revenue
  Q1 A   250   100
      B   150   250
      C    100   300
  Q2 A   150   200
      B   300   175
      C   220   225
```

```python
>>> df.le(df_multindex, level=1)  # doctest: +SKIP
    cost  revenue
  Q1 A  True  True
      B  True  True
      C  True  True
  Q2 A  False  True
      B  True  False
      C  True  False
```

`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 \( \text{npartitions} \) 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** *(\text{axis=None, skipna=True, split_every=False})*

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

This docstring was copied from pandas.core.frame.DataFrame.idxmax. Some inconsistencies with the Dask version may exist.

**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** *(\text{axis=None, skipna=True, split_every=False})*

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

This docstring was copied from pandas.core.frame.DataFrame.idxmin. Some inconsistencies with the Dask version may exist.

**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.

iloc
Purely integer-location based indexing for selection by position.
Only indexing the column positions is supported. Trying to select row positions will raise a ValueError.
See Indexing into Dask DataFrames for more.

Examples

```python
>>> df.iloc[:, [2, 0, 1]]  # doctest: +SKIP
```

index
Return dask Index instance

info (buf=None, verbose=False, memory_usage=False)
Concise summary of a Dask DataFrame.

isin (values)
Whether each element in the DataFrame is contained in values.
This docstring was copied from pandas.core.frame.DataFrame.isin.
Some inconsistencies with the Dask version may exist.

Parameters

values [iterable, Series, DataFrame or dict] 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 dict, 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 DataFrame of booleans showing whether each element in the DataFrame is
contained in values.

See also:

DataFrame.eq Equality test for DataFrame.
Series.isin Equivalent method on Series.
Series.str.contains Test if pattern or regex is contained within a string of a Series or Index.

Examples

```python
>>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
index=['falcon', 'dog'])
>>> df  # doctest: +SKIP
   num_legs  num_wings
falcon      2           2
  dog        4           0
```

When values is a list check whether every value in the DataFrame is present in the list (which animals
have 0 or 2 legs or wings)
>>> df.isin([0, 2])  # doctest: +SKIP
    num_legs num_wings
falcon   True   True
dog      False  True

When `values` is a dict, we can pass values to check for each column separately:

>>> df.isin({'num_wings': [0, 3]})  # doctest: +SKIP
    num_legs num_wings
falcon   False  False
dog      False  True

When `values` is a Series or DataFrame the index and column must match. Note that ‘falcon’ does not match based on the number of legs in df2.

>>> other = pd.DataFrame({'num_legs': [8, 2],'num_wings': [0, 2]},
                      index=['spider', 'falcon'])

>>> df.isin(other)  # doctest: +SKIP
    num_legs num_wings
falcon   True   True
dog      False  False

`isna()`

Detect missing values.

This docstring was copied from pandas.core.frame.DataFrame.isna.

Some inconsistencies with the Dask version may exist.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Returns

- **DataFrame** Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

- `DataFrame.isnull` Alias of `isna`.
- `DataFrame.notna` Boolean inverse of `isna`.
- `DataFrame.dropna` Omit axes labels with missing values.
- `isna` Top-level `isna`.

Examples

Show which entries in a DataFrame are NA.

```python
>>> df = pd.DataFrame({'age': [5, 6, np.nan],
                             pd.Timestamp('1940-04-25')],
                    'name': ['Alfred', 'Batman', ''],
                    'toy': [None, 'Batmobile', 'Joker']})
```
Show which entries in a DataFrame are NA.

>>> df = pd.DataFrame({'age': [5.0, 6.0, np.nan], 'born': [1939-05-27, 1940-04-25, np.nan], 'name': ['Batman', 'Joker'], 'toy': [Batmobile, None]})
# doctest: +SKIP

>>> df
age     born       name       toy
0      5.0    NaT         Alfred    None
1      6.0  1939-05-27  Batman    Batmobile
2      NaN    NaT         Joker      

>>> df.isna()  # doctest: +SKIP
age     born       name       toy
0  False   True     False     True
1  False   False    False     False
2  True    False    False     False

Show which entries in a Series are NA.

>>> ser = pd.Series([5, 6, np.nan])
# doctest: +SKIP

>>> ser
0  5.0
1  6.0
2  NaN
dtype: float64

>>> ser.isna()  # doctest: +SKIP
0  False
1  False
2  True
dtype: bool

isnull()

Detect missing values.

This docstring was copied from pandas.core.frame.DataFrame.isnull.

Some inconsistencies with the Dask version may exist.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.
NaN, get mapped to True values. Everything else gets mapped to False values. Characters such as empty
strings ' ' or numpy.inf are not considered NA values (unless you set pandas.options.mode.
use_inf_as_na = True).

Returns

DataFrame  Mask of bool values for each element in DataFrame that indicates whether
an element is not an NA value.

See also:

DataFrame.isnull  Alias of isna.
DataFrame.notna  Boolean inverse of isna.
DataFrame.dropna  Omit axes labels with missing values.
isna  Top-level isna.

Examples

Show which entries in a DataFrame are NA.

4.9. DataFrame  467
```python
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})

>>> df
age    born       name      toy
0    5.0  NaT  Alfred   None
1    6.0 1939-05-27  Batman  Batmobile
2   NaN 1940-04-25   Joker     

>>> df.isna()
age    born       name      toy
0    False  True  False     True
1    False   False  False     False
2    True   False  False     False

```

Show which entries in a Series are NA.

```python
>>> ser = pd.Series([5, 6, np.NaN])

>>> ser
0   5.0
1   6.0
2  NaN
dtype: float64

>>> ser.isna()
0  False
1  False
2  True

dtype: bool
```

`iterrows()`
Iterate over DataFrame rows as (index, Series) pairs.

This docstring was copied from pandas.core.frame.DataFrame.iterrows.

Some inconsistencies with the Dask version may exist.

Yields

- `index` [label or tuple of label] The index of the row. A tuple for a `MultiIndex`.
- `data` [Series] The data of the row as a Series.
- `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,
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.

```
int 1.0
float 1.5
Name: 0, dtype: float64
```

```
print(row['int'].dtype)  # doctest: +SKIP
float64
```

```
print(df['int'].dtype)  # doctest: +SKIP
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`.

**itertuples** *(index=True, name='Pandas')*

Iterate over DataFrame rows as namedtuples.

This docstring was copied from pandas.core.frame.DataFrame.itertuples.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **index** [bool, default True] If True, return the index as the first element of the tuple.
- **name** [str or None, default “Pandas”] The name of the returned namedtuples or None to return regular tuples.

**Yields**

collections.namedtuple  Yields a namedtuple for each row in the DataFrame with the first field possibly being the index and following fields being the column values.

**See also:**

- **DataFrame.iterrows** Iterate over DataFrame rows as (index, Series) pairs.
- **DataFrame.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**

```python
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]})  # doctest: +SKIP
...  index=['dog', 'hawk'])
```

```
dog  num_legs  num_wings
     4          0
```

(continues on next page)
By setting the `index` parameter to `False` we can remove the index as the first element of the tuple:

```
>>> for row in df.itertuples(index=False):  # doctest: +SKIP
...    print(row)
...
Pandas(num_legs=4, num_wings=0)
Pandas(num_legs=2, num_wings=2)
```

With the `name` parameter set we set a custom name for the yielded namedtuples:

```
>>> for row in df.itertuples(name='Animal'):  # doctest: +SKIP
...    print(row)
...
Animal(Index='dog', num_legs=4, num_wings=0)
Animal(Index='hawk', num_legs=2, num_wings=2)
```

`join` (other, on=None, how='left', lsuffix='', rsuffix='', npartitions=None, shuffle=None)

Join columns of another DataFrame.

This docstring was copied from pandas.core.frame.DataFrame.join.

Some inconsistencies with the Dask version may exist.

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, 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** [str, list of str, or array-like, optional] Column or index level name(s) in the caller to join on the index in `other`, otherwise joins index-on-index. If multiple values given, the `other` DataFrame must have a MultiIndex. Can pass an array as the join key if it is 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`’s index.
  - outer: form union of calling frame’s index (or column if on is specified) with `other`’s index, and sort it. lexicographically.
  - inner: form intersection of calling frame’s index (or column if on is specified) with `other`’s index, preserving the order of the calling’s one.

- **lsuffix** [str, default ‘’] Suffix to use from left frame’s overlapping columns.

- **rsuffix** [str, default ‘’] Suffix to use from right frame’s overlapping columns.
sort [bool, default False (Not supported in Dask)] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword).

Returns

DataFrame A dataframe containing columns from both the caller and other.

See also:

DataFrame.merge For column(s)-on-column(s) operations.

Notes

Parameters on, lsuffix, and rsuffix are not supported when passing a list of DataFrame objects.

Support for specifying index levels as the on parameter was added in version 0.23.0.

Examples

```python
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
                      'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})

>>> df # doctest: +SKIP
   key  A
0    K0  A0
1    K1  A1
2    K2  A2
3    K3  A3
4    K4  A4
5    K5  A5

>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
                          'B': ['B0', 'B1', 'B2']})

>>> other # doctest: +SKIP
   key  B
0    K0  B0
1    K1  B1
2    K2  B2

Join DataFrames using their indexes.

>>> df.join(other, lsuffix='_caller', rsuffix='_other') # doctest: +SKIP
   key_changed  A  key_changed  B
0        K0    A0        K0    B0
1        K1    A1        K1    B1
2        K2    A2        K2    B2
3        K3    A3        NaN   NaN
4        K4    A4        NaN   NaN
5        K5    A5        NaN   NaN
```

If we want to join using the key columns, we need to set key to be the index in both df and other. The joined DataFrame will have key as its index.
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 df. This method preserves the original DataFrame’s index in the result.

```python
>>> df.join(other.set_index('key'), on='key')  # doctest: +SKIP

A       B
0  K0  A0  B0
1  K1  A1  B1
2  K2  A2  B2
3  K3  A3  NaN
4  K4  A4  NaN
5  K5  A5  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.

This docstring was copied from pandas.core.frame.DataFrame.last.

Some inconsistencies with the Dask version may exist.

Parameters

offset [string, DateOffset, dateutil.relativedelta]

Returns

subset [same type as caller]

Raises

TypeError If the index is not a DatetimeIndex

See also:

first Select initial periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')  # doctest: +SKIP
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)  # doctest: +SKIP
>>> ts  # doctest: +SKIP
               A
2018-04-09  1
```

(continues on next page)
Get the rows for the last 3 days:

```python
>>> ts.last('3D')  # doctest: +SKIP
A
2018-04-13 3
2018-04-15 4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

**le** *(other, axis='columns', level=None)*

Less than or equal to of dataframe and other, element-wise (binary operator *le*).

Among flexible wrappers (*eq, ne, le, lt, ge, gt*) to comparison operators.

Equivalent to *==, !=, <=, <, >=, >* with support to choose axis (rows or columns) and level for comparison.

**Parameters**

*other* [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

*axis* [{0 or ‘index’, 1 or ‘columns’}, default ‘columns’] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).

*level* [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**

*DataFrame of bool* Result of the comparison.

**See also:**

*DataFrame.eq* Compare DataFrames for equality elementwise.

*DataFrame.ne* Compare DataFrames for inequality elementwise.

*DataFrame.le* Compare DataFrames for less than inequality or equality elementwise.

*DataFrame.lt* Compare DataFrames for strictly less than inequality elementwise.

*DataFrame.ge* Compare DataFrames for greater than inequality or equality elementwise.

*DataFrame.gt* Compare DataFrames for strictly greater than inequality elementwise.

**Notes**

Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN != NaN*).

**Examples**
Comparison with a scalar, using either the operator or method:

```python
>>> df == 100  # doctest: +SKIP
    cost  revenue
A  False   True
B  False  False
C  True  False
```

```python
>>> df.eq(100)  # doctest: +SKIP
    cost  revenue
A  False   True
B  False  False
C  True  False
```

When `other` is a `Series`, the columns of a DataFrame are aligned with the index of `other` and broadcast:

```python
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])  # doctest: +SKIP
    cost  revenue
A  True   True
B  True  False
C  False   True
```

Use the method to control the broadcast axis:

```python
>>> df.ne(pd.Series([100, 300], index=['A', 'D']), axis='index')  # doctest: +SKIP
    cost  revenue
A  True  False
B  False  True
C  True  True
D  True  True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in `other`:

```python
>>> df == [250, 100]  # doctest: +SKIP
    cost  revenue
A  True   True
B  False  False
C  False  False
```

Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')  # doctest: +SKIP
    cost  revenue
A  True  False
B  False  True
C  True  False
```
Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150],  # doctest: +SKIP...
                        index=['A', 'B', 'C', 'D'])
>>> other  # doctest: +SKIP
   revenue
A   300
B   250
C   100
D   150
```

```python
>>> df.gt(other)  # doctest: +SKIP
    cost  revenue
A  False  False
B  False  False
C  False   True
D  False  False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],  # doctest: +SKIP...
                              'revenue': [100, 250, 300, 200, 175, 225],
                              index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
                              ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex  # doctest: +SKIP
    cost  revenue
Q1 A   250   100
   B   150   250
   C   100   300
Q2 A   150   200
   B   300   175
   C   220   225
```

```python
>>> df.le(df_multindex, level=1)  # doctest: +SKIP
    cost  revenue
Q1 A  True  True
   B  True  True
   C  True  True
Q2 A False  True
   B  True False
   C  True False
```

`loc`

Purely label-location based indexer for selection by label.

```python
>>> df.loc["b"]  # doctest: +SKIP
>>> df.loc["b":"d"]  # doctest: +SKIP
```

`lt` (other, axis='columns', level=None)

Less than of dataframe and other, element-wise (binary operator lt).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters
other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
axis [{0 or ‘index’, 1 or ‘columns’}, default ‘columns’] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).
level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.
DataFrame.ne Compare DataFrames for inequality elementwise.
DataFrame.ile Compare DataFrames for less than inequality or equality elementwise.
DataFrame.ilt Compare DataFrames for strictly less than inequality elementwise.
DataFrame.ige Compare DataFrames for greater than inequality or equality elementwise.
DataFrame.igt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100],
                     'revenue': [100, 250, 300]},
                    index=['A', 'B', 'C'])
```

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100  # doctest: +SKIP
cost  revenue
A    False   True
B    False   False
C     True   False
```

```python
>>> df.eq(100)  # doctest: +SKIP
cost  revenue
A    False   True
B    False   False
C     True   False
```

When other is a Series, the columns of a DataFrame are aligned with the index of other and broadcast:
Use the method to control the broadcast axis:

```python
>>> df.ne(pd.Series([100, 300], index=['A', 'D']), axis='index')  # doctest: +SKIP
   cost  revenue
A  True  False
B  True  True
C  True  True
D  True  True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in `other`:

```python
>>> df == [250, 100]  # doctest: +SKIP
   cost  revenue
A  True  True
B  False False
C  False False
```

Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')  # doctest: +SKIP
   cost  revenue
A  True  False
B  False True
C  True  False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},  # doctest: +SKIP...
                        index=['A', 'B', 'C', 'D'])
>>> other  # doctest: +SKIP
  revenue
A   300
B   250
C   100
D   150
```

```python
>>> df.gt(other)  # doctest: +SKIP
   cost  revenue
A  False  False
B  False  False
C  False  True
D  False  False
```

Compare to a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],  # ...
                               ...
                               'revenue': [100, 250, 300, 200, 175, 225]},
                              ...
                              index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2']])
```

(continues on next page)
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:

```python
>>> 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()`:

```python
>>> ddf.compute()
    x  y
0  1  1.0
1  2  2.0
2  4  3.0
3  7  4.0
4 11  5.0
```

```python
>>> 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 prepend/appending that many rows, depending on sign:

```python
>>> def diff(df, periods=1):
...     before, after = (periods, 0) if periods > 0 else (0, -periods)
...     return df.map_overlap(lambda df, periods=1: df.diff(periods),
...                            periods=periods)
...
>>> ddf.map_overlap(lambda df: df.rolling('2D').sum(),
...                  pd.Timedelta('2D'), 0).compute()
2017-01-01 0.0
```

If you have a `DatetimeIndex`, you can use a `pd.Timedelta` for time-based windows.

```python
>>> 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(),
...                  '2D', 0).compute()
```

(continues on next page)
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. Arguments and keywords may contain `Scalar`, `Delayed` or regular python objects. DataFrame-like args (both dask and pandas) will be repartitioned to align (if necessary) before applying the 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`.

Examples

Given a DataFrame, Series, or Index, such as:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> ddf.map_partitions(func).clear_divisions()  # doctest: +SKIP
```

`mask (cond, other=nan)`

Replace values where the condition is True.

This docstring was copied from pandas.core.frame.DataFrame.mask.

Some inconsistencies with the Dask version may exist.

**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 (Not supported in Dask)] Whether to perform the operation in place on the data.

- `axis` [int, default None (Not supported in Dask)] Alignment axis if needed.

- `level` [int, default None (Not supported in Dask)] Alignment level if needed.
errors [str, {'raise', 'ignore'}, default raise (Not supported in Dask)] Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

• raise : allow exceptions to be raised.

• ignore : suppress exceptions. On error return original object.

try_cast [boolean, default False (Not supported in Dask)] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True (Not supported in Dask)] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use errors.

Returns

wh [same type as caller]

See also:

DataFrame.where() Return an object of same shape as self.

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

```python
>>> s = pd.Series(range(5))  # doctest: +SKIP
>>> s.where(s > 0)  # doctest: +SKIP
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

```python
>>> s.mask(s > 0)  # doctest: +SKIP
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

```python
>>> s.where(s > 1, 10)  # doctest: +SKIP
0    10
1    10
2     2
```

(continues on next page)
3 3
dtype: int64

```python
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])  # doctest: +SKIP
>>> m = df % 3 == 0  # doctest: +SKIP
>>> df.where(m, -df)  # doctest: +SKIP
A  B
0 -1
1  3
2 -5
3 -7
4  9
>>> df.where(m, -df) == np.where(m, df, -df)  # doctest: +SKIP
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)  # doctest: +SKIP
A  B
0 True True
1 True True
2 True True
3 True True
4 True True
```

```python
max (axis=None, skipna=True, split_every=False, out=None)
```
Return the maximum of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.max.

Some inconsistencies with the Dask version may exist.

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)]) Axis for the function to be applied on.
- `skipna` [bool, default True] Exclude NA/null values when computing the result.
- `level` [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- `numeric_only` [bool, default None (Not supported in Dask)] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- **`kwargs`** Additional keyword arguments to be passed to the function.

Returns

- `max` [Series or DataFrame (if level specified)]

See also:
Series.sum Return the sum.
Series.min Return the minimum.
Series.max Return the maximum.
Series.idxmin Return the index of the minimum.
Series.idxmax Return the index of the maximum.
DataFrame.min Return the sum over the requested axis.
DataFrame.min Return the minimum over the requested axis.
DataFrame.max Return the maximum over the requested axis.
DataFrame.idxmin Return the index of the minimum over the requested axis.
DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```python
>>> idx = pd.MultiIndex.from_arrays([  # doctest: +SKIP
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)  # doctest: +SKIP
>>> s
blooded animal
warm   dog   4
      falcon  2
cold   fish   0
      spider  8
Name: legs, dtype: int64

>>> s.max()  # doctest: +SKIP
8

Max using level names, as well as indices.

```python
>>> s.max(level='blooded')  # doctest: +SKIP
blooded
warm   4
      cold  8
Name: legs, dtype: int64
```python

```python
>>> s.max(level=0)  # doctest: +SKIP
blooded
warm   4
      cold  8
Name: legs, dtype: int64
```python

mean (axis=None, skipna=True, split_every=False, dtype=None, out=None)
Return the mean of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.mean.

Some inconsistencies with the Dask version may exist.

Parameters
axis [[index (0), columns (1)]] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None (Not supported in Dask)] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

mean [Series or DataFrame (if level specified)]

memory_usage (index=True, deep=False)

Return the memory usage of each column in bytes.

This docstring was copied from pandas.core.frame.DataFrame.memory_usage.

Some inconsistencies with the Dask version may exist.

The memory usage can optionally include the contribution of the index and elements of object dtype.

This value is displayed in DataFrame.info by default. This can be suppressed by setting pandas.options.display.memory_usage to False.

Parameters

index [bool, default True] Specifies whether to include the memory usage of the DataFrame’s index in returned Series. If index=True the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating object dtypes for system-level memory consumption, and include it in the returned values.

Returns

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

See also:

numpy.ndarray.nbytes Total bytes consumed by the elements of an ndarray.

Series.memory_usage Bytes consumed by a Series.

pandas.Categorical Memory-efficient array for string values with many repeated values.

DataFrame.info Concise summary of a DataFrame.

Examples

```python
dl = [int64, float64, complex128, object, bool] # doctest: +SKIP

data = dict(((t, np.ones(shape=5000).astype(t)) for t in dtypes)) # doctest: +SKIP

df = pd.DataFrame(data) # doctest: +SKIP

df.head() # doctest: +SKIP
```

(continues on next page)
The memory footprint of `object` dtype columns is ignored by default:

```
>>> df.memory_usage(index=False) # doctest: +SKIP
int64 40000
float64 40000
complex128 80000
object 40000
bool 5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True) # doctest: +SKIP
5168
```

```
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)
```

This will merge the two datasets, either on the indices, a certain column in each dataset or the index in one dataset and the column in another.

### Parameters

- **right:** dask.dataframe.DataFrame
- **how:** ([`‘left’, ‘right’, ‘outer’, ‘inner’`], default: `‘inner’`) 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
on [label or list] Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column to join on in the left DataFrame. Other than in pandas arrays and lists are only support if their length is 1.

right_on [label or list, or array-like] Column to join on in the right DataFrame. Other than in pandas arrays and lists are only support if their length is 1.

left_index [boolean, default False] Use the index from the left DataFrame as the join key.

right_index [boolean, default False] Use the index from the right DataFrame as the join key.

suffixes [2-length sequence (tuple, list, …)] Suffix to apply to overlapping column names in the left and right side, respectively.

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.

npartitions: int, None, or 'auto' The ideal number of output partitions. This is only utilised when performing a hash_join (merging on columns only). If None npartitions = max(lhs.npartitions, rhs.npartitions)

shuffle: {'disk', 'tasks'}, optional Either 'disk' for single-node operation or 'tasks' for distributed operation. Will be inferred by your current scheduler.

Notes

There are three ways to join dataframes:

1. Joining on indices. In this case the divisions are aligned using the function dask.dataframe.multi.align_partitions. Afterwards, each partition is merged with the pandas merge function.

2. Joining one on index and one on column. In this case the divisions of dataframe merged by index \(d_i\) are used to divide the column merged dataframe \(d_c\) one using dask.dataframe.multi.rearrange_by_divisions. In this case the merged dataframe \(d_m\) has the exact same divisions as \(d_i\). This can lead to issues if you merge multiple rows from \(d_c\) to one row in \(d_i\).

3. Joining both on columns. In this case a hash join is performed using dask.dataframe.multi.hash_join.

min (axis=None, skipna=True, split_every=False, out=None) Return the minimum of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.min.

Some inconsistencies with the Dask version may exist.

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)]] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None (Not supported in Dask)] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

min [Series or DataFrame (if level specified)]

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.min Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```python
>>> idx = pd.MultiIndex.from_arrays([              # doctest: +SKIP
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)  # doctest: +SKIP
>>> s  # doctest: +SKIP
blooded animal
  warm   dog   4
     falcon   2
cold   fish   0
         spider   8
Name: legs, dtype: int64

>>> s.min()  # doctest: +SKIP
0
```

Min using level names, as well as indices.
mod \( (\text{other}, \text{axis}='\text{columns}', \text{level}=\text{None}, \text{fill_value}=\text{None}) \)

Modulo of dataframe and other, element-wise (binary operator \texttt{mod}).

Equivalent to \texttt{dataframe \% other}, but with support to substitute a \texttt{fill_value} for missing data in one of the inputs. With reverse version, \texttt{rmod}.

Among flexible wrappers (\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{mod}, \texttt{pow}) to arithmetic operators: +, −, *, /, //, %, **.

Parameters

- \texttt{other} [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- \texttt{axis} [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- \texttt{level} [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- \texttt{fill_value} [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

- \texttt{DataFrame} Result of the arithmetic operation.

See also:

- \texttt{DataFrame.add} Add DataFrames.
- \texttt{DataFrame.sub} Subtract DataFrames.
- \texttt{DataFrame.mul} Multiply DataFrames.
- \texttt{DataFrame.div} Divide DataFrames (float division).
- \texttt{DataFrame.truediv} Divide DataFrames (float division).
- \texttt{DataFrame.floordiv} Divide DataFrames (integer division).
- \texttt{DataFrame.mod} Calculate modulo (remainder after division).
- \texttt{DataFrame.pow} Calculate exponential power.

Notes

Mismatched indices will be unioned together.
Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
...                     'degrees': [360, 180, 360],
...                     'index': ['circle', 'triangle', 'rectangle'])

>>> df  # doctest: +SKIP
angles  degrees
circle   0       360
triangle 3       180
rectangle 4       360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
angles  degrees
circle   1       361
triangle 4       181
rectangle 5       361
```

```python
>>> df.add(1)  # doctest: +SKIP
angles  degrees
circle   1       361
triangle 4       181
rectangle 5       361
```

Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle   0.0     36.0
triangle 0.3     18.0
rectangle 0.4    36.0
```

```python
>>> df.rdiv(10)  # doctest: +SKIP
angles  degrees
circle   inf    0.027778
triangle 3.333333 0.055556
rectangle 2.500000 0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
angles  degrees
circle   -1      358
triangle   2      178
rectangle   3      358
```

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
angles  degrees
circle   -1      358
triangle   2      178
rectangle   3      358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),  # doctest: +SKIP
...         axis='index')
```

(continues on next page)
Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
                      index=['circle', 'triangle', 'rectangle'])

>>> other
angles
circle 0
triangle 3
rectangle 4
```

```
>>> df * other
angles degrees
circle 0 NaN
triangle 9 NaN
rectangle 16 NaN
```

```
>>> df.mul(other, fill_value=0)
angles degrees
circle 0 0.0
triangle 9 0.0
rectangle 16 0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
                            'degrees': [360, 180, 360, 360, 540, 720],
                            'index': [['A', 'A', 'A', 'B', 'B', 'B'],
                                  ['circle', 'triangle', 'rectangle',
                                   'square', 'pentagon', 'hexagon']])

>>> df_multindex
angles  degrees
A circle   0  360
triangle   3  180
rectangle  4  360
B square   4  360
pentagon   5  540
hexagon    6  720

>>> df.div(df_multindex, level=1, fill_value=0)
angles  degrees
A circle   NaN  1.0
triangle   1.0  1.0
rectangle  1.0  1.0
B square   0.0  0.0
pentagon   0.0  0.0
hexagon    0.0  0.0
```

```
mul (other, axis='columns', level=None, fill_value=None)
Multiplication of dataframe and other, element-wise (binary operator mul).
```

4.9. DataFrame 491
Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, `rmul`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.

**Parameters**

- `other` [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- `axis` [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- `level` [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- `fill_value` [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

- `DataFrame` Result of the arithmetic operation.

See also:

- `DataFrame.add` Add DataFrames.
- `DataFrame.sub` Subtract DataFrames.
- `DataFrame.mul` Multiply DataFrames.
- `DataFrame.div` Divide DataFrames (float division).
- `DataFrame.truediv` Divide DataFrames (float division).
- `DataFrame.floordiv` Divide DataFrames (integer division).
- `DataFrame.mod` Calculate modulo (remainder after division).
- `DataFrame.pow` Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4], # doctest: +SKIP
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'])
>>> df  # doctest: +SKIP
angles  degrees
circle   0       360
triangle 3       180
rectangle 4       360
```

Add a scalar with operator version which return the same results.
>>> df + 1  # doctest: +SKIP
angles  degrees
circle   1   361
triangle  4   181
rectangle 5   361

>>> df.add(1)  # doctest: +SKIP
angles  degrees
circle   1   361
triangle  4   181
rectangle 5   361

Divide by constant with reverse version.

>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle  0.0  36.0
triangle 0.3  18.0
rectangle 0.4  36.0

>>> df.rdiv(10)  # doctest: +SKIP
angles  degrees
circle  inf  0.027778
triangle 3.333333 0.055556
rectangle 2.500000 0.027778

Subtract a list and Series by axis with operator version.

>>> df - [1, 2]  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle   2  178
rectangle  3  358

>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle   2  178
rectangle  3  358

>>> other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])

Multiply a DataFrame of different shape with operator version.

>>> df - other  # doctest: +SKIP
angles  degrees
circle  0  359
triangle -2  179
rectangle -3  359

(continues on next page)
triangle 3
rectangle 4

```python
>>> df * other  # doctest: +SKIP
   angles  degrees
  circle   0    NaN
   triangle 9    NaN
  rectangle 16   NaN
```

```python
>>> df.mul(other, fill_value=0)  # doctest: +SKIP
   angles  degrees
  circle   0    0.0
   triangle 9    0.0
  rectangle 16   0.0
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
                                'degrees': [360, 180, 360, 360, 540, 720],
                                'index': [['A', 'A', 'A', 'B', 'B', 'B'],
                                          ['circle', 'triangle', 'rectangle',
                                           'square', 'pentagon', 'hexagon']])
```

```python
>>> df_multindex  # doctest: +SKIP
  angles  degrees
A  circle   0    360
   triangle 3    180
  rectangle 4    360
B  square   4    360
   pentagon 5    540
  hexagon  6    720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)  # doctest: +SKIP
   angles  degrees
A  circle   NaN   1.0
   triangle 1.0   1.0
  rectangle 1.0   1.0
B  square   0.0   0.0
   pentagon 0.0   0.0
  hexagon  0.0   0.0
```

`ndim`

Return dimensionality

`ne` *(other, axis='columns', level=None)*

Not equal to of dataframe and other, element-wise (binary operator \(\ne\)).

Among flexible wrappers (*eq, ne, le, lt, ge, gt*) to comparison operators.

Equivalent to \(==, \neq, \leq, <, \geq, >\) with support to choose axis (rows or columns) and level for comparison.

**Parameters**

- `other` [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
axis [0 or ‘index’, 1 or ‘columns’, default ‘columns’] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’).

level [int or label] Broadcast across a level, matching Index values on the passed Multi-Index level.

Returns

DataFrame of bool Result of the comparison.

See also:

DataFrame.eq Compare DataFrames for equality elementwise.
DataFrame.ne Compare DataFrames for inequality elementwise.
DataFrame.le Compare DataFrames for less than inequality or equality elementwise.
DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.
DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.
DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

```python
>>> df = pd.DataFrame({'cost': [250, 150, 100],  # doctest: +SKIP
                    'revenue': [100, 250, 300]},
                    index=['A', 'B', 'C'])
>>> df
    cost  revenue
A    250      100
B    150      250
C    100      300

Comparison with a scalar, using either the operator or method:

```python
>>> df == 100  # doctest: +SKIP
    cost  revenue
A    False    True
B    False    False
C     True    False
```  

```python
>>> df.eq(100)  # doctest: +SKIP
    cost  revenue
A    False    True
B    False    False
C     True    False
```  

When other is a Series, the columns of a DataFrame are aligned with the index of other and broadcast:

```python
>>> df != pd.Series([100, 250], index=['cost', 'revenue'])  # doctest: +SKIP
    cost  revenue
A     True    True
```  

(continues on next page)
Use the method to control the broadcast axis:

```python
>>> df.ne(pd.Series([100, 300], index=['A', 'D']), axis='index')  # doctest: +SKIP
    cost  revenue
A   True     False
B   True      True
C   True      True
D   True      True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in `other`:

```python
>>> df == [250, 100]  # doctest: +SKIP
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```python
>>> df.eq([250, 250, 100], axis='index')  # doctest: +SKIP
    cost  revenue
A  False     False
B  False    False
C   True     False
D  False    False
```

Compare to a DataFrame of different shape.

```python
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},  # doctest: +SKIP
                      index=['A', 'B', 'C', 'D'])
>>> other  # doctest: +SKIP
   revenue
A   300
B   250
C   100
D   150
```

```python
>>> df.gt(other)  # doctest: +SKIP
    cost  revenue
A  False     False
B  False    False
C   True     False
D  False    False
```

Compare to a MultiIndex by level:

```python
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],  # doctest: +SKIP
                               'revenue': [100, 250, 300, 200, 175, 225]},
                              index=['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'])
>>> df_multindex  # doctest: +SKIP
```

(continues on next page)
nlargest \((n=5, columns=None, split_every=None)\)

Return the first \(n\) rows ordered by \(columns\) in descending order.

This docstring was copied from pandas.core.frame.DataFrame.nlargest.

Some inconsistencies with the Dask version may exist.

Return the first \(n\) rows with the largest values in \(columns\), in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to \(df.sort_values(columns, ascending=False).head(n)\), but more performant.

**Parameters**

- \(n\) [int] Number of rows to return.
- \(columns\) [label or list of labels] Column label(s) to order by.
- \(keep\) [[‘first’, ‘last’, ‘all’], default ‘first’ (Not supported in Dask)] Where there are duplicate values:
  - \(first\) : prioritize the first occurrence(s)
  - \(last\) : prioritize the last occurrence(s)
  - \(all\) [do not drop any duplicates, even it means] selecting more than \(n\) items.

New in version 0.24.0.

**Returns**

- DataFrame The first \(n\) rows ordered by the given columns in descending order.

**See also:**

- DataFrame.nsmallest Return the first \(n\) rows ordered by \(columns\) in ascending order.
- DataFrame.sort_values Sort DataFrame by the values.
- DataFrame.head Return the first \(n\) rows without re-ordering.
Notes

This function cannot be used with all column types. For example, when specifying columns with `object` or `category` dtypes, `TypeError` is raised.

Examples

```python
>>> df = pd.DataFrame({'population': [59000000, 65000000, 4340000, #
... 434000, 434000, 337000, 11300,
... 11300, 11300],
... 'GDP': [1937894, 2583560, 12011, 4520, 12128,
... 17036, 182, 31, 311],
... 'alpha-2': ['IT', 'FR', 'MT', 'MV', 'BN',
... 'IS', 'NR', 'TV', 'AI']},
... index=['Italy', 'France', 'Malta',
... 'Maldives', 'Brunei', 'Iceland',
... 'Nauru', 'Tuvalu', 'Anguilla'])
>>> df  # doctest: +SKIP
   population     GDP  alpha-2
Italy     59000000 1937894    IT
France    65000000 2583560    FR
Malta     4340000  12011     MT
Maldives  4340000  4520      MV
Brunei    4340000  12128     BN
Iceland   3370000  17036     IS
Nauru     11300     182      NR
Tuvalu    11300     31       TV
Anguilla  11300     311      AI
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column “population”.

```python
>>> df.nlargest(3, 'population')  # doctest: +SKIP
   population     GDP  alpha-2
France    65000000 2583560    FR
Italy     59000000 1937894    IT
Malta     4340000  12011     MT
```

When using `keep='last'`, ties are resolved in reverse order:

```python
>>> df.nlargest(3, 'population', keep='last')  # doctest: +SKIP
   population     GDP  alpha-2
France    65000000 2583560    FR
Italy     59000000 1937894    IT
Brunei    4340000  12128     BN
```

When using `keep='all'`, all duplicate items are maintained:

```python
>>> df.nlargest(3, 'population', keep='all')  # doctest: +SKIP
   population     GDP  alpha-2
France    65000000 2583560    FR
Italy     59000000 1937894    IT
Malta     4340000  12011     MT
Maldives  4340000  4520      MV
Brunei    4340000  12128     BN
```
To order by the largest values in column “population” and then “GDP”, we can specify multiple columns like in the next example.

```python
>>> df.nlargest(3, ['population', 'GDP'])  # doctest: +SKIP
    population     GDP    alpha-2
  France  65000000  2583560  FR
  Italy   59000000  1937894  IT
 Brunei  4340000   12128   BN
```

`notnull()`
Detect existing (non-missing) values.

This docstring was copied from pandas.core.frame.DataFrame.notnull.

Some inconsistencies with the Dask version may exist.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings ‘’ or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

**Returns**

- `DataFrame`: Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

**See also:**

- `DataFrame.notnull`: Alias of notna.
- `DataFrame.isna`: Boolean inverse of notna.
- `DataFrame.dropna`: Omit axes labels with missing values.
- `notna`: Top-level notna.

**Examples**

Show which entries in a DataFrame are not NA.

```python
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],  # doctest: +SKIP
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
```

```python
>>> df  # doctest: +SKIP
    age  born      name    toy
0  5.0   NaT  Alfred   None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25   Joker
```

```python
>>> df.notna()  # doctest: +SKIP
    age  born      name    toy
0  True  False   True  False
1  True   True   True  True
2  False  True   True  True
```

Show which entries in a Series are not NA.

```python
```python
>>> ser = pd.Series([5, 6, np.NaN])  # doctest: +SKIP
>>> ser  # doctest: +SKIP
0   5.0
1   6.0
2  NaN
dtype: float64
```

```python
>>> ser.notna()  # doctest: +SKIP
0   True
1   True
2  False
dtype: bool
```

npartitions

Return number of partitions

nsmallest (n=5, columns=None, split_every=None)

Return the first n rows ordered by columns in ascending order.

This docstring was copied from pandas.core.frame.DataFrame.nsmallest.

Some inconsistencies with the Dask version may exist.

Return the first n rows with the smallest values in columns, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to df.sort_values(columns, ascending=True).head(n), but more performant.

**Parameters**

- **n** [int] Number of items to retrieve.
- **columns** [list or str] Column name or names to order by.
- **keep** [‘first’, ‘last’, ‘all’], default ‘first’ (Not supported in Dask)] Where there are duplicate values:
  - first: take the first occurrence.
  - last: take the last occurrence.
  - all: do not drop any duplicates, even it means selecting more than n items.

New in version 0.24.0.

**Returns**

DataFrame

See also:

- **DataFrame.nlargest** Return the first n rows ordered by columns in descending order.
- **DataFrame.sort_values** Sort DataFrame by the values.
- **DataFrame.head** Return the first n rows without re-ordering.

**Examples**
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000, 434000, 434000, 434000, 337000, 11300, 11300, 11300, 11300], 'GDP': [1937894, 2583560, 12011, 4520, 12128, 17036, 182, 38, 311], 'alpha-2': ['IT', 'FR', 'MT', 'MV', 'BN', 'IS', 'NR', 'TV', 'AI']}, index=['Italy', 'France', 'Malta', 'Maldives', 'Brunei', 'Iceland', 'Nauru', 'Tuvalu', 'Anguilla'])

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column “a”.

```python
>>> df.nsmallest(3, 'population')
# doctest: +SKIP

<table>
<thead>
<tr>
<th></th>
<th>population</th>
<th>GDP</th>
<th>alpha-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nauru</td>
<td>11300</td>
<td>182</td>
<td>NR</td>
</tr>
<tr>
<td>Tuvalu</td>
<td>11300</td>
<td>38</td>
<td>TV</td>
</tr>
<tr>
<td>Anguilla</td>
<td>11300</td>
<td>311</td>
<td>AI</td>
</tr>
</tbody>
</table>
```

When using `keep='last'`, ties are resolved in reverse order:

```python
>>> df.nsmallest(3, 'population', keep='last')
# doctest: +SKIP

<table>
<thead>
<tr>
<th></th>
<th>population</th>
<th>GDP</th>
<th>alpha-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anguilla</td>
<td>11300</td>
<td>311</td>
<td>AI</td>
</tr>
<tr>
<td>Tuvalu</td>
<td>11300</td>
<td>38</td>
<td>TV</td>
</tr>
<tr>
<td>Nauru</td>
<td>11300</td>
<td>182</td>
<td>NR</td>
</tr>
</tbody>
</table>
```

When using `keep='all'`, all duplicate items are maintained:

```python
>>> df.nsmallest(3, 'population', keep='all')
# doctest: +SKIP

<table>
<thead>
<tr>
<th></th>
<th>population</th>
<th>GDP</th>
<th>alpha-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nauru</td>
<td>11300</td>
<td>182</td>
<td>NR</td>
</tr>
<tr>
<td>Tuvalu</td>
<td>11300</td>
<td>38</td>
<td>TV</td>
</tr>
<tr>
<td>Anguilla</td>
<td>11300</td>
<td>311</td>
<td>AI</td>
</tr>
</tbody>
</table>
```

To order by the largest values in column “a” and then “c”, we can specify multiple columns like in the next example.

```python
>>> df.nsmallest(3, ['population', 'GDP'])
# doctest: +SKIP

<table>
<thead>
<tr>
<th></th>
<th>population</th>
<th>GDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuvalu</td>
<td>11300</td>
<td>38</td>
</tr>
<tr>
<td>Nauru</td>
<td>11300</td>
<td>182</td>
</tr>
<tr>
<td>Anguilla</td>
<td>11300</td>
<td>311</td>
</tr>
</tbody>
</table>
```

`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

**partitions**

Slice dataframe by partitions

This allows partitionwise slicing of a Dask Dataframe. You can perform normal Numpy-style slicing but now rather than slice elements of the array you slice along partitions so, for example, `df.partitions[:5]` produces a new Dask Dataframe of the first five partitions.

**Returns**

- A Dask DataFrame

**Examples**

```python
def partitions[0]  # doctest: +SKIP
def partitions[:3]  # doctest: +SKIP
def partitions[:10]  # doctest: +SKIP```

**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.

The action of function differs significantly depending on the active task scheduler. If the task scheduler supports asynchronous computing, such as is the case of the dask.distributed scheduler, then persist will return immediately and the return value’s task graph will contain Dask Future objects. However if the task scheduler only supports blocking computation then the call to persist will block and the return value’s task graph will contain concrete Python results.

This function is particularly useful when using distributed systems, because the results will be kept in distributed memory, rather than returned to the local process as with compute.

**Parameters**

- **scheduler** [string, optional] Which scheduler to use like “threads”, “synchronous” or “processes”. 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 function.

**Returns**

- New dask collections backed by in-memory data
See also:

dask.base.persist

pipe (func, *args, **kwargs)
Apply func(self, *args, **kwargs).
This docstring was copied from pandas.core.frame.DataFrame.pipe.
Some inconsistencies with the Dask version may exist.

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:

DataFrame.apply, DataFrame.applymap, Series.map

Notes

Use .pipe when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead
of writing

```python
>>> f(g(h(df), arg1=a), arg2=b, arg3=c) # doctest: +SKIP
```
You can write

```python
>>> (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:

```python
>>> (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. With reverse version, rpow.
Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other  [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
axis  [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
level  [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
fill_value  [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame  Result of the arithmetic operation.

See also:

DataFrame.add  Add DataFrames.
DataFrame.sub  Subtract DataFrames.
DataFrame.mul  Multiply DataFrames.
DataFrame.div  Divide DataFrames (float division).
DataFrame.truediv  Divide DataFrames (float division).
DataFrame.floordiv  Divide DataFrames (integer division).
DataFrame.mod  Calculate modulo (remainder after division).
DataFrame.pow  Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
                      ... 'degrees': [360, 180, 360],
                      ...     index=['circle', 'triangle', 'rectangle'])
>>> df  # doctest: +SKIP
(continues on next page)```
Add a scalar with operator version which return the same results.

```plaintext
>>> df + 1  # doctest: +SKIP
      angles   degrees
    circle     1   361
    triangle    4   181
    rectangle   5   361
```

Divide by constant with reverse version.

```plaintext
>>> df.div(10)  # doctest: +SKIP
     angles   degrees
   circle  0.00    36.00
   triangle  0.30    18.00
  rectangle  0.40    36.00
```

Subtract a list and Series by axis with operator version.

```plaintext
>>> df - [1, 2]  # doctest: +SKIP
      angles   degrees
    circle    -1   358
    triangle     2   178
    rectangle    3   358
```

```plaintext
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
     angles   degrees
   circle    -1   358
   triangle     2   178
  rectangle    3   358
```

Multiply a DataFrame of different shape with operator version.
```
>>> other = pd.DataFrame({'angles': [0, 3, 4]}, # doctest: +SKIP
    index=['circle', 'triangle', 'rectangle'])
```
```
>>> df * other # doctest: +SKIP
    angles    degrees
    circle   0          NaN
    triangle 3          NaN
    rectangle 4          NaN
```
```
>>> df.mul(other, fill_value=0) # doctest: +SKIP
    angles    degrees
    circle   0       0.0
    triangle 9       0.0
    rectangle 16     0.0
```

Divide by a MultiIndex by level.
```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6], # doctest:
    'degrees': [360, 180, 360, 360, 540, 720]},
    index=[['A', 'A', 'A', 'B', 'B', 'B'],
            ['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon']])
```
```
>>> df_multindex
    angles    degrees
    A circle   0       360
    triangle   3       180
    rectangle  4       360
    B square   4       360
    pentagon   5       540
    hexagon    6       720
```
```
>>> df.div(df_multindex, level=1, fill_value=0) # doctest: +SKIP
    angles    degrees
    A circle   NaN      1.0
    triangle   1.0      1.0
    rectangle  1.0      1.0
    B square   0.0      0.0
    pentagon   0.0      0.0
    hexagon    0.0      0.0
```

**prod** *(axis=None, skipna=True, split_every=False, dtype=None, out=None, min_count=None)*

Return the product of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.prod.

Some inconsistencies with the Dask version may exist.

**Parameters**

- `axis` *(index (0), columns (1))* Axis for the function to be applied on.
- `skipna` *[bool, default True]* Exclude NA/null values when computing the result.
level [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None (Not supported in Dask)] 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 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

**kwargs Additional keyword arguments to be passed to the function.

Returns

prod [Series or DataFrame (if level specified)]

Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()  # doctest: +SKIP
1.0
```

This can be controlled with the min_count parameter

```
>>> pd.Series([]).prod(min_count=1)  # doctest: +SKIP
nan
```

Thanks to the skipna parameter, min_count handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()  # doctest: +SKIP
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)  # doctest: +SKIP
nan
```

quantile (q=0.5, axis=0, method='default')
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

method [[‘default’, ‘tdigest’, ‘dask’], optional] What method to use. By default will use dask’s internal custom algorithm (‘dask’). If set to ‘tdigest’ will use tdigest for floats and ints and fallback to the ‘dask’ otherwise.

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

```python
import numexpr
numexpr.set_nthreads(1)
```

See also:

pdbas.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. With reverse version, `add`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed Multi-Index level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

Dataframe Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.
Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4], 'degrees': [360, 180, 360]},
                   index=['circle', 'triangle', 'rectangle'])
```

```
angles  degrees
circle   0       360
triangle 3       180
rectangle 4       360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
angles  degrees
circle   1       361
triangle 4       181
rectangle 5       361
```

```python
>>> df.add(1)  # doctest: +SKIP
angles  degrees
circle   1       361
triangle 4       181
rectangle 5       361
```

Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle   0.0     36.0
triangle 0.3     18.0
rectangle 0.4    36.0
```

```python
>>> df.rdiv(10)  # doctest: +SKIP
angles  degrees
circle  inf   0.027778
triangle 3.333333 0.055556
rectangle 2.500000 0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
angles  degrees
circle   -1      358
triangle   2      178
rectangle   3      358
```

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
angles  degrees
circle   -1      358
triangle   2      178
rectangle   3      358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
         axis='index')  # doctest: +SKIP
```

(continues on next page)
Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4],
                        'degrees': [360, 180, 360, 360, 540, 720]},
                        index=['A', 'A', 'A', 'B', 'B', 'B'],
                        columns=['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])
```

```python
>>> df * other
```

```python
>>> df.mul(other, fill_value=0)
```

Divide by a MultiIndex by level.

```python
>>> df_div = df.div(df_multindex, level=1, fill_value=0)
```

Random split:

```python
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

```python
>>> a, b = df.random_split([0.5, 0.5]) # doctest: +SKIP
```

80/10/10 split, consistent random_state

```python
>>> a, b, c = df.random_split([0.8, 0.1, 0.1], random_state=123) # doctest: +SKIP
```

**rdiv** (other, axis='columns', level=None, fill_value=None)
Floating division of dataframe and other, element-wise (binary operator /).
Equivalent to other / dataframe, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, truediv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
axis [[0 or ‘index’, 1 or ‘columns‘]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns‘). For Series input, axis to match Series index on.
level [int or label] Broadcast across a level, matching Index values on the passed Multi-Index level.
fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
**DataFrame.mod** Calculate modulo (remainder after division).

**DataFrame.pow** Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
... 'degrees': [360, 180, 360]},
... index=['circle', 'triangle', 'rectangle'])

>>> df
   angles  degrees
circle    0      360
triangle   3      180
rectangle  4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
   angles  degrees
circle    1      361
triangle   4      181
rectangle  5      361
```

Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
   angles  degrees
circle  0.0      36.0
triangle 0.3      18.0
rectangle 0.4     36.0
```

```python
>>> df.rdiv(10)  # doctest: +SKIP
   angles  degrees
circle  inf      0.027778
triangle 3.333333 0.055556
rectangle 2.500000 0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
   angles  degrees
circle   -1      358
triangle    2      178
rectangle    3      358
```
Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4]},
                     index=['circle', 'triangle', 'rectangle'])
other
```

```plaintext
angles
circle 0
triangle 3
rectangle 4
```

```python
df * other  # doctest: +SKIP
```

```plaintext
angles      degrees
circle  0  NaN
triangle  9  NaN
rectangle 16  NaN
```

```python
df.mul(other, fill_value=0)  # doctest: +SKIP
```

```plaintext
angles     degrees
circle    0.0
triangle  0.0
rectangle 0.0
```

Divide by a MultiIndex by level.

```python
df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
                             'degrees': [360, 180, 360, 360, 540, 720]},
                            index=['A', 'A', 'A', 'B', 'B', 'B'],
                            columns=['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])
```

```python
df_multindex  # doctest: +SKIP
```

```plaintext
angles     degrees
A circle   0.0  360
A triangle 3.0  180
A rectangle 4.0  360
B square   4.0  360
B pentagon 5.0  540
B hexagon  6.0  720
```

```python
df.div(df_multindex, level=1, fill_value=0)  # doctest: +SKIP
```

```plaintext
angles     degrees
A circle   NaN  1.0
(continues on next page)
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

```python
>>> 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:

```python
>>> 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).

```python
>>> 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:

```python
>>> def sum_and_count(x):
...     return pd.Series({'count': x.count(), 'sum': x.sum()},
                      columns=['count', 'sum'])

>>> 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.

```python
>>> def sum_and_count(x):
...     return pd.DataFrame({'count': x.count(), 'sum': x.sum()},
                          columns=['count', 'sum'])

>>> 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.

This docstring was copied from pandas.core.frame.DataFrame.rename. Some inconsistencies with the Dask version may exist.
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 (Not supported in Dask)] 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 (Not supported in Dask)] Also copy underlying data

- **inplace** [boolean, default False (Not supported in Dask)] Whether to return a new DataFrame. If True then value of copy is ignored.

- **level** [int or level name, default None (Not supported in Dask)] In case of a MultiIndex, only rename labels in the specified level.

**Returns**

- **renamed** [DataFrame]

**See also:**

pandas.DataFrame.rename_axis

**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.

```python
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})  # doctest: +SKIP
>>> df.rename(index=str, columns={"A": "a", "B": "c"})  # doctest: +SKIP
              a  c
0   1   4
1   2   5
2   3   6
```

```python
>>> df.rename(index=str, columns={"A": "a", "C": "c"})  # doctest: +SKIP
              a  B
0   1   4
1   2   5
2   3   6
```

```
>>> df.rename(index=str, columns={"C": "c"})  # doctest: +SKIP
```

Using axis-style parameters

```python
>>> df.rename(str.lower, axis='columns')  # doctest: +SKIP
 a   b
0   1   4
1   2   5
2   3   6
```
```python
>>> df.rename({1: 2, 2: 4}, axis='index')  # doctest: +SKIP
   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. 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**

```python
>>> df = df.repartition(npartitions=10)  # doctest: +SKIP
>>> df = df.repartition(divisions=[0, 5, 10, 20])  # doctest: +SKIP
>>> df = df.repartition(freq='7d')  # doctest: +SKIP
```

**replace** (*to_replace=None, value=None, regex=False*)

Replace values given in *to_replace* with *value*.

This docstring was copied from pandas.core.frame.DataFrame.replace.

Some inconsistencies with the Dask version may exist.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with .loc or .iloc, which require you to specify a location to update with some value.

**Parameters**

- **to_replace** [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.
  - numeric, str or regex:
    - numeric: numeric values equal to *to_replace* will be replaced with *value*
    - str: string exactly matching *to_replace* will be replaced with *value*
    - regex: regexes matching *to_replace* will be replaced with *value*
  - list of str, regex, or numeric:
    - First, if *to_replace* and *value* are both lists, they must be the same length.
    - Second, if regex=True then all of the strings in both lists will be interpreted as regexes otherwise they will match directly. This doesn’t matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.

4.9. DataFrame
dict:

- Dicts can be used to specify different replacement values for different existing values. For example, (\{a\': \ 'b', \ 'y\': \ 'z'\}) replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the value parameter should be None.

- For a DataFrame a dict can specify that different values should be replaced in different columns. For example, (\{'a\': 1, \ 'b\': \ 'z'\}) looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in value. The value parameter should not be None in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.

- For a DataFrame nested dictionaries, e.g., (\{'a\': \{\ 'b\': \ np.nan\}\}), are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The value parameter should be None to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) cannot be regular expressions.

None:

- This means that the regex argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If value is also None then this must be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching to_replace with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [bool, default False (Not supported in Dask)] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None (Not supported in Dask)] Maximum size gap to forward or backward fill.

regex [bool or same types as to_replace, default False] Whether to interpret to_replace and/or value as regular expressions. If this is True then to_replace must be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case to_replace must be None.

method [\{'pad', \ 'ffill', \ 'bfill', None\}] (Not supported in Dask)] The method to use when for replacement, when to_replace is a scalar, list or tuple and value is None.

Changed in version 0.23.0: Added to DataFrame.

Returns

DataFrame Object after replacement.

Raises

AssertionError

- If regex is not a bool and to_replace is not None.

TypeError

- If to_replace is a dict and value is not a list, dict, ndarray, or Series.

- If value is also None then this must be a nested dictionary or Series.
• If `to_replace` is `None` and `regex` is not compilable into a regular expression or is a list, dict, ndarray, or Series.

• When replacing multiple `bool` or `datetime64` objects and the arguments to `to_replace` does not match the type of the value being replaced

**ValueError**

• If a list or an ndarray is passed to `to_replace` and `value` but they are not the same length.

See also:

*DataFrame.fillna* Fill NA values.

*DataFrame.where* Replace values based on boolean condition.

*Series.str.replace* Simple string replacement.

**Notes**

• Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.

• Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers are strings, then you can do this.

• This method has a lot of options. You are encouraged to experiment and play with this method to gain intuition about how it works.

• When dict is used as the `to_replace` value, it is like key(s) in the dict are the `to_replace` part and value(s) in the dict are the `value` parameter.

**Examples**

**Scalar ‘to_replace‘ and ‘value‘**

```
>>> s = pd.Series([0, 1, 2, 3, 4])  # doctest: +SKIP
>>> s.replace(0, 5)  # doctest: +SKIP
0 5
1 1
2 2
3 3
4 4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],  # doctest: +SKIP
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
```

```
>>> df.replace(0, 5)  # doctest: +SKIP
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```
List-like ‘to_replace’

```python
>>> df.replace([0, 1, 2, 3], 4)  # doctest: +SKIP
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```python
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])  # doctest: +SKIP
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```python
>>> s.replace([1, 2], method='bfill')  # doctest: +SKIP
0  0
1  3
2  3
3  3
4  4
dtype: int64
```

dict-like ‘to_replace’

```python
>>> df.replace({0: 10, 1: 100})  # doctest: +SKIP
   A  B  C
0 10  5  a
1 100  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

```python
>>> df.replace({'A': 0, 'B': 5}, 100)  # doctest: +SKIP
   A  B  C
0 100  100  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

```python
>>> df.replace({'A': {0: 100, 4: 400}})  # doctest: +SKIP
   A  B  C
0 100  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4 400  9  e
```

Regular expression ‘to_replace’

```python
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'], # doctest: +SKIP
                    'B': ['abc', 'bar', 'xyz']})
```
(continues on next page)
Note that when replacing multiple bool or datetime64 objects, the data types in the `to_replace` parameter must match the data type of the value being replaced:

```python
>>> df = pd.DataFrame({'A': [True, False, True],  # doctest: +SKIP 
                     ...  
                     'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises  # doctest: +SKIP
Traceback (most recent call last):
  ...  
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a TypeError because one of the dict keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the `to_replace` parameter:

```python
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])  # doctest: +SKIP
>>> s.replace({'a': None})  # doctest: +SKIP
0    10
Name: a, dtype: int64
```

When one uses a dict as the `to_replace` value, it is like the value(s) in the dict are equal to the `value` parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None):

```python
>>> s.replace({'a': None})  # doctest: +SKIP
0    10
Name: a, dtype: int64
```
When `value=None` and `to_replace` is a scalar, list or tuple, `replace` uses the method parameter (default 'pad') to do the replacement. So this is why the ‘a’ values are being replaced by 10 in rows 1 and 2 and ‘b’ in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```python
>>> s.replace('a', None)  # doctest: +SKIP
0 10
1 10
2 10
3 b
4 b
dtype: object
```

`resample` (rule, closed=None, label=None)

Resample time-series data.

This docstring was copied from pandas.core.frame.DataFrame.resample.

Some inconsistencies with the Dask version may exist.

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** [str] The offset string or object representing target conversion.
- **how** [str (Not supported in Dask)] Method for down/re-sampling, default to ‘mean’ for downsampling.

  Deprecated since version 0.18.0: The new syntax is `.resample(...).mean()`, or `.resample(...).apply(<func>)`

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0 (Not supported in Dask)] Which axis to use for up- or down-sampling. For Series this will default to 0, i.e. along the rows. Must be `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`.
- **fill_method** [str, default None (Not supported in Dask)] Filling method for upsampling.

  Deprecated since version 0.18.0: The new syntax is `.resample(...).<func>()`, e.g. `.resample(...).pad()`


- **label** [{‘right’, ‘left’}, default None] 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’}, default ‘start’ (Not supported in Dask)] For `PeriodIndex` only, controls whether to use the start or end of `rule`. 
kind [‘timestamp’, ‘period’], optional, default None (Not supported in Dask)] Pass ‘timestamp’ to convert the resulting index to a DateTimeIndex or ‘period’ to convert it to a PeriodIndex. By default the input representation is retained.

loffset [timedelta, default None (Not supported in Dask)] Adjust the resampled time labels.

limit [int, default None (Not supported in Dask)] Maximum size gap when reindexing with fill_method.

Deprecated since version 0.18.0.

base [int, default 0 (Not supported in Dask)] 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 [str, optional (Not supported in Dask)] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [str or int, optional (Not supported in Dask)] For a MultiIndex, level (name or number) to use for resampling. level must be datetime-like.

New in version 0.19.0.

Returns

Resampler object

See also:

groupby Group by mapping, function, label, or list of labels.

Series.resample Resample a Series.

DataFrame.resample Resample a DataFrame.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```python
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')  # doctest: +SKIP
>>> series = pd.Series(range(9), index=index)  # doctest: +SKIP
>>> 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
```

(continues on next page)
Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()  # doctest: +SKIP
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()  # doctest: +SKIP
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()  # doctest: +SKIP
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 # doctest: +SKIP
2000-01-01 00:00:00    0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    1
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the NaN values using the pad method.

```
>>> series.resample('30S').pad()[0:5]  # doctest: +SKIP
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.
Pass a custom function via apply

```python
>>> def custom_resampler(array_like): # doctest: +SKIP
...    return np.sum(array_like) + 5
... >>> series.resample('3T').apply(custom_resampler) # doctest: +SKIP
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`.

Resample a year by quarter using ‘start’ `convention`. Values are assigned to the first quarter of the period.

```python
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01', # doctest: +SKIP
...   freq='A',
...   periods=2))
>>> s # doctest: +SKIP
2012 1
2013 2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq() # doctest: +SKIP
2012Q1 1.0
2012Q2 NaN
2012Q3 NaN
2012Q4 NaN
2013Q1 2.0
2013Q2 NaN
2013Q3 NaN
2013Q4 NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using ‘end’ `convention`. Values are assigned to the last month of the period.

```python
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01', # doctest: +SKIP
...   freq='Q',
...   periods=4))
>>> q # doctest: +SKIP
2018Q1 1
2018Q2 2
2018Q3 3
2018Q4 4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq() # doctest: +SKIP
2018-03 1.0
2018-04 NaN
```

(continues on next page)
For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```python
>>> d = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],  # doctest: +SKIP
...            'volume': [50, 60, 40, 100, 50, 100, 40, 50]})

>>> df = pd.DataFrame(d)  # doctest: +SKIP

>>> df['week_starting'] = pd.date_range('01/01/2018',  # doctest: +SKIP
...                                       periods=8,
...                                       freq='W')

>>> df  # doctest: +SKIP
   price  volume week_starting
0      10       50   2018-01-07
1      11       60   2018-01-14
2       9       40   2018-01-21
3      13      100   2018-01-28
4      14       50   2018-02-04
5      18      100   2018-02-11
6      17       40   2018-02-18
7      19       50   2018-02-25

>>> df.resample('M', on='week_starting').mean()  # doctest: +SKIP
   price  volume
2018-01-31  10.75   62.5
2018-02-28  17.00   60.0
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on which level the resampling needs to take place.

```python
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')  # doctest: +SKIP

>>> d2 = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],  # doctest: +SKIP
...            'volume': [50, 60, 40, 100, 50, 100, 40, 50]})

>>> df2 = pd.DataFrame(d2,  # doctest: +SKIP
...                     index=pd.MultiIndex.from_product([days,
...                                                           ['morning',
...                                                            'afternoon']]))

>>> df2  # doctest: +SKIP
          price  volume
2000-01-01 morning     10       50
                      afternoon   11       60
2000-01-02 morning      9       40
                      afternoon   13      100
2000-01-03 morning     14       50
                      afternoon   18      100
2000-01-04 morning     17       40
                      afternoon   19       50
```

(continues on next page)
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. With reverse version, floordiv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- axis [{0 or ‘index’, 1 or ‘columns’}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

- DataFrame.add Add DataFrames.
- DataFrame.sub Subtract DataFrames.
- DataFrame.mul Multiply DataFrames.
- DataFrame.div Divide DataFrames (float division).
- DataFrame.truediv Divide DataFrames (float division).
**DataFrame.floordiv**  Divide DataFrames (integer division).

**DataFrame.mod**  Calculate modulo (remainder after division).

**DataFrame.pow**  Calculate exponential power.

### Notes

Mismatched indices will be unioned together.

### Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4], # doctest: +SKIP
...                   'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df  # doctest: +SKIP
     angles  degrees
    circle     0   360
    triangle    3   180
    rectangle   4   360

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
     angles  degrees
    circle    1   361
    triangle   4   181
    rectangle   5   361
``` 

```python
>>> df.add(1)  # doctest: +SKIP
     angles  degrees
    circle    1   361
    triangle   4   181
    rectangle   5   361
```

Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
     angles  degrees
    circle 0.0   36.0
    triangle 0.3   18.0
    rectangle 0.4   36.0
``` 

```python
>>> df.rdiv(10)  # doctest: +SKIP
     angles  degrees
    circle inf   0.027778
    triangle 3.333333   0.055556
    rectangle 2.500000   0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
     angles  degrees
    circle   -1   358
    triangle    2   178
    rectangle    3   358
```
Multiply a DataFrame of different shape with operator version.

```python
doctest: +SKIP
other = pd.DataFrame({'angles': [0, 3, 4]},
                      index=['circle', 'triangle', 'rectangle'])
doctest: +SKIP
```

```python
doctest: +SKIP
other
```

```python
doctest: +SKIP
angles
circle 0
triangle 3
rectangle 4
```

```python
doctest: +SKIP
df * other
```

```python
doctest: +SKIP
angles
circle 0 NaN
triangle 9 NaN
rectangle 16 NaN
```

```python
doctest: +SKIP
df.mul(other, fill_value=0)
```

```python
doctest: +SKIP
angles
circle 0 0.0
triangle 9 0.0
rectangle 16 0.0
```

Divide by a MultiIndex by level.

```python
doctest: +SKIP
df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6], 'degrees': [360, 180, 360, 540, 720],
                              index=['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])
doctest: +SKIP
```

```python
doctest: +SKIP
df_multindex
```

```python
doctest: +SKIP
angles
A circle 0 360
triangle 3 180
rectangle 4 360
B square 4 360
pentagon 5 540
hexagon 6 720
```

```python
doctest: +SKIP
df.div(df_multindex, level=1, fill_value=0)
```

```python
doctest: +SKIP
angles
A circle NaN 1.0
```

(continues on next page)
\textbf{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. With reverse version, mod.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [(0 or 'index', 1 or 'columns')] Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

- **DataFrame** Result of the arithmetic operation.

**See also:**

- \texttt{DataFrame.add} Add DataFrames.
- \texttt{DataFrame.sub} Subtract DataFrames.
- \texttt{DataFrame.mul} Multiply DataFrames.
- \texttt{DataFrame.div} Divide DataFrames (float division).
- \texttt{DataFrame.truediv} Divide DataFrames (float division).
- \texttt{DataFrame.floordiv} Divide DataFrames (integer division).
- \texttt{DataFrame.mod} Calculate modulo (remainder after division).
- \texttt{DataFrame.pow} Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**
>>> df = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
...                     'degrees': [360, 180, 360],
...                     'index': ['circle', 'triangle', 'rectangle']})

Add a scalar with operator version which return the same results.

>>> df + 1  # doctest: +SKIP
angles  degrees
circle  1  361
triangle  4  181
rectangle  5  361

>>> df.add(1)  # doctest: +SKIP
angles  degrees
circle  1  361
triangle  4  181
rectangle  5  361

Divide by constant with reverse version.

>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle  0  36.0
triangle  0.3  18.0
rectangle  0.4  36.0

>>> df.rdiv(10)  # doctest: +SKIP
angles  degrees
circle  inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778

Subtract a list and Series by axis with operator version.

>>> df - [1, 2]  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle  2  178
rectangle  3  358

>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle  2  178
rectangle  3  358

>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),  # doctest: +SKIP
...         axis='index')
angles  degrees
circle  -1  359

(continues on next page)
Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4]}, # doctest: +SKIP
                     index=['circle', 'triangle', 'rectangle'])
other # doctest: +SKIP
angles
circle 0
triangle 3
de
rectangle 4

df * other # doctest: +SKIP
angles degrees
circle 0 NaN
de
triangle 9 NaN
de
rectangle 16 NaN

df.mul(other, fill_value=0) # doctest: +SKIP
angles degrees
circle 0 0.0
de
triangle 9 0.0
de
rectangle 16 0.0

Divide by a MultiIndex by level.

df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6], # doctest: +SKIP
                            'degrees': [360, 180, 360, 360, 540, 720],
                            index=['A', 'A', 'A', 'B', 'B', 'B'],
                            ['circle', 'triangle', 'rectangle',
                             'square', 'pentagon', 'hexagon'])

df_multindex # doctest: +SKIP
angles degrees
A circle 0 360
de
triangle 3 180
de
rectangle 4 360
B square 4 360
pentagon 5 540
hexagon 6 720

df.div(df_multindex, level=1, fill_value=0) # doctest: +SKIP
angles degrees
A circle NaN 1.0
de
triangle 1.0 1.0
de
rectangle 1.0 1.0
B square 0.0 0.0
pentagon 0.0 0.0
hexagon 0.0 0.0
```

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. With reverse version, mul.
Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.

axis [[0 or ‘index’, 1 or ‘columns’]] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.

level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4], # doctest: +SKIP
...                     'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'])
>>> df # doctest: +SKIP
   angles  degrees
circle    0       360
triangle  3       180
rectangle 4       360
```

Add a scalar with operator version which return the same results.
Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
dest degrees
circle 0.0 36.0
triangle 0.3 18.0
rectangle 0.4 36.0
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
dest degrees
circle -1 358
triangle 2 178
rectangle 3 358
```

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
dest degrees
circle -1 358
triangle 2 178
rectangle 3 358
```

Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])
other  # doctest: +SKIP
```

(continues on next page)
triangle  3
rectangle  4

```python
>>> df * other  # doctest: +SKIP
   angles  degrees
  circle    0    NaN
   triangle   9    NaN
  rectangle  16    NaN
```

```python
>>> df.mul(other, fill_value=0)  # doctest: +SKIP
   angles  degrees
  circle    0    0.0
   triangle   9    0.0
  rectangle  16    0.0
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],  # doctest: ...
                              ...
              'degrees': [360, 180, 360, 360, 540, 720],
              ...
            index=['A', 'A', 'A', 'B', 'B', 'B'],
            ...
      ['circle', 'triangle', 'rectangle',
       'square', 'pentagon', 'hexagon'])
```

```python
>>> df_multindex  # doctest: +SKIP
      angles  degrees
     A  circle    0  360
       triangle   3  180
      rectangle   4  360
     B  square    4  360
      pentagon    5  540
    hexagon    6  720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0)  # doctest: +SKIP
   angles  degrees
  A circle     NaN   1.0
   triangle    1.0   1.0
  rectangle    1.0   1.0
  B square    0.0    0.0
    pentagon  0.0    0.0
   hexagon    0.0    0.0
```

`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.

This docstring was copied from pandas.core.frame.DataFrame.round.

Some inconsistencies with the Dask version may exist.

**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

**See also:**

numpy.around, Series.round

**Examples**

```python
>>> df = pd.DataFrame(np.random.random([3, 3]),
...                   columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
```

```python
>>> df # doctest: +SKIP
      A    B    C
first  0.028208 0.992815 0.173891
second 0.038683 0.645646 0.577595
third  0.877076 0.149370 0.491027
```

```python
>>> df.round(2) # doctest: +SKIP
      A    B   C
first 0.03 0.99 0.17
second 0.04 0.65 0.58
third 0.88 0.15 0.49
```

```python
>>> df.round({'A': 1, 'C': 2}) # doctest: +SKIP
      A  B   C
first 0.0 0.99 0.17
second 0.0 0.65 0.58
third 0.9 0.15 0.49
```

```python
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C']) # doctest: +SKIP
>>> df.round(decimals) # doctest: +SKIP
      A   B    C
first 0.0 0.99 0.17
second 0.0 0.65 0.58
third 0.9 0.15 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. With reverse version, pow.
Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

other [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
axis [{0 or ‘index’, 1 or ‘columns’}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
level [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.
fill_value [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

DataFrame.add Add DataFrames.
DataFrame.sub Subtract DataFrames.
DataFrame.mul Multiply DataFrames.
DataFrame.div Divide DataFrames (float division).
DataFrame.truediv Divide DataFrames (float division).
DataFrame.floordiv Divide DataFrames (integer division).
DataFrame.mod Calculate modulo (remainder after division).
DataFrame.pow Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples
Add a scalar with operator version which return the same results.

```
>>> df + 1  # doctest: +SKIP
angles  degrees
circle  1    361
triangle 5    181
rectangle 5    361
```

Divide by constant with reverse version.

```
>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle  0.0  36.0
triangle 0.3  18.0
rectangle 0.4  36.0
```

```
>>> df.rdiv(10)  # doctest: +SKIP
angles  degrees
circle  inf  0.027778
triangle 3.333333 0.055556
rectangle 2.500000 0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle  2  178
rectangle  3  358
```

```
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle  2  178
rectangle  3  358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),  
  axis='index')  # doctest: +SKIP
angles  degrees
circle  -1  359
```
Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4], # doctest: +SKIP
...                       index=['circle', 'triangle', 'rectangle'])
>>> other # doctest: +SKIP
angles
circle 0
triangle 3
rectangle 4

>>> df * other # doctest: +SKIP
angles  degrees
circle 0.0 NaN
triangle 9.0 NaN
rectangle 16.0 NaN
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6], # doctest: +SKIP
...                             'degrees': [360, 180, 360, 360, 540, 720],
...                             index=['A', 'A', 'A', 'B', 'B', 'B'],
...                             ['circle', 'triangle', 'rectangle',
...                             'square', 'pentagon', 'hexagon'])
>>> df_multindex # doctest: +SKIP
angles  degrees
A circle 0 360
triangle 3 180
rectangle 4 360
B square 4 360
pentagon 5 540
hexagon 6 720
```

```python
>>> df.div(df_multindex, level=1, fill_value=0) # doctest: +SKIP
angles  degrees
A circle NaN 1.0
triangle 1.0 1.0
rectangle 1.0 1.0
B square 0.0 0.0
pentagon 0.0 0.0
hexagon 0.0 0.0
```

**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. With reverse version, *sub*.

4.9. Dataframe 539
Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [(0 or ‘index’, 1 or ‘columns’)] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** [int or label] Broadcast across a level, matching Index values on the passed Multi-Index level.
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

- **DataFrame** Result of the arithmetic operation.

**See also:**

- `DataFrame.add` Add DataFrames.
- `DataFrame.sub` Subtract DataFrames.
- `DataFrame.mul` Multiply DataFrames.
- `DataFrame.div` Divide DataFrames (float division).
- `DataFrame.truediv` Divide DataFrames (float division).
- `DataFrame.floordiv` Divide DataFrames (integer division).
- `DataFrame.mod` Calculate modulo (remainder after division).
- `DataFrame.pow` Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4], 'degrees': [360, 180, 360]})
>>> df
   angles  degrees
circle    0     360
triangle  3     180
rectangle 4     360
```

Add a scalar with operator version which return the same results.
Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
   angles  degrees
circle  0.0     36.0
triangle 0.3   18.0
rectangle 0.4  36.0
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
   angles  degrees
circle  -1     358
triangle   2   178
rectangle  3   358
```

Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])
other  # doctest: +SKIP
   angles
circle  0
```

(continues on next page)
triangle 3
rectangle 4

>>> df * other # doctest: +SKIP
angles degrees
circle 0 NaN
triangle 9 NaN
rectangle 16 NaN

>>> df.mul(other, fill_value=0) # doctest: +SKIP
angles degrees
circle 0 0.0
triangle 9 0.0
rectangle 16 0.0

Divide by a MultiIndex by level.

>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
                           'degrees': [360, 180, 360, 360, 540, 720]},
                           index=['A', 'A', 'A', 'B', 'B', 'B'],
                           columns=['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])

>>> df_multindex # doctest: +SKIP
angles degrees
A circle 0 360
triangle 3 180
rectangle 4 360
B square 4 360
pentagon 5 540
hexagon 6 720

>>> df.div(df_multindex, level=1, fill_value=0) # doctest: +SKIP
angles degrees
A circle NaN 1.0
triangle 1.0 1.0
rectangle 1.0 1.0
B square 0.0 0.0
pentagon 0.0 0.0
hexagon 0.0 0.0

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. With reverse version, `truediv`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- **axis** [{0 or ‘index’, 1 or ‘columns’}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
level [int or label] Broadcast across a level, matching Index values on the passed Multi-
Index level.

fill_value [float or None, default None] Fill existing missing (NaN) values, and any new
element needed for successful DataFrame alignment, with this value before computa-
tion. If data in both corresponding DataFrame locations is missing the result will be
missing.

Returns

DataFrame Result of the arithmetic operation.

See also:

Dataframe.add Add DataFrames.
Dataframe.sub Subtract DataFrames.
Dataframe.mul Multiply DataFrames.
Dataframe.div Divide DataFrames (float division).
Dataframe.truediv Divide DataFrames (float division).
Dataframe.floordiv Divide DataFrames (integer division).
Dataframe.mod Calculate modulo (remainder after division).

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4], # doctest: +SKIP
                      'degrees': [360, 180, 360]},
                     index=['circle', 'triangle', 'rectangle'])
>>> df
          angles  degrees
circle     0      360
triangle    3      180
rectangle   4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
          angles  degrees
    circle     1      361
    triangle    4      181
    rectangle   5      361
```

```python
>>> df.add(1)  # doctest: +SKIP
          angles  degrees
    circle     1      361
    triangle    4      181
    rectangle   5      361
```
Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle  0.0  36.0
triangle 0.3  18.0
rectangle 0.4  36.0
```

```python
>>> df.rdiv(10)  # doctest: +SKIP
angles  degrees
circle   inf  0.027778
triangle 3.333333  0.055556
rectangle 2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle 2  178
rectangle 3  358
```

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
angles  degrees
circle  -1  358
triangle 2  178
rectangle 3  358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), axis='index')  # doctest: +SKIP
angles  degrees
circle  -1  359
triangle 2  179
rectangle 3  359
```

Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])
```

```python
>>> df * other  # doctest: +SKIP
angles  degrees
circle  0  NaN
triangle 9  NaN
rectangle 16  NaN
```

```python
>>> df.mul(other, fill_value=0)  # doctest: +SKIP
angles  degrees
circle  0  0.0
triangle 9  0.0
rectangle 16  0.0
```
Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame(
...     {'angles': [0, 3, 4, 4, 5, 6],
...      # doctest: +SKIP
...     'degrees': [360, 180, 360, 360, 540, 720],
...     'index': [['A', 'A', 'A', 'B', 'B', 'B'],
...               ['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon']])

>>> df_multindex  # doctest: +SKIP
angles   degrees
A circle  0     360
triangle  3     180
rectangle 4     360
B square  4     360
pentagon  5     540
hexagon  6     720

>>> df.div(df_multindex, level=1, fill_value=0)  # doctest: +SKIP
angles   degrees
A circle  NaN    1.0
triangle  1.0    1.0
rectangle 1.0    1.0
B square  0.0    0.0
pentagon  0.0    0.0
hexagon  0.0    0.0
```

**sample** *(n=None, frac=None, replace=False, random_state=None)*

Random sample of items

**Parameters**

- **n** [int, optional] Number of items to return is not supported by dask. Use frac instead.
- **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 the DataFrame’s columns based on the column dtypes.

This docstring was copied from pandas.core.frame.DataFrame.select_dtypes.

Some inconsistencies with the Dask version may exist.

**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 `np.number` or `'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 'datetime64[ns, tz]'`

Examples

```python
>>> df = pd.DataFrame({'a': [1, 2] * 3,  # doctest: +SKIP
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
a  b  c
0  1  True  1.0
1  2  False  2.0
2  1  True  1.0
3  2  False  2.0
4  1  True  1.0
5  2  False  2.0
```

```python
>>> df.select_dtypes(include='bool')  # doctest: +SKIP
b
0  True
1  False
2  True
3  False
4  True
5  False
```

```python
>>> df.select_dtypes(include=['float64'])  # doctest: +SKIP
c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```
sem\((axis=None, skipna=None, ddof=1, split_every=False)\)

Return unbiased standard error of the mean over requested axis.

This docstring was copied from pandas.core.frame.DataFrame.sem.

Some inconsistencies with the Dask version may exist.

Normalized by N-1 by default. This can be changed using the ddof argument

**Parameters**

- **axis**
  
  \[\{\text{index (0), columns (1)}\}\]

- **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 (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

- **ddof**
  
  [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

- **numeric_only**
  
  [boolean, default None (Not supported in Dask)] 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)]

**set_index\(\text{(other, drop=True, sorted=False, npartitions=None, divisions=None, inplace=False, **kwargs)}\)**

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 https://docs.dask.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.

**inplace** [bool, optional] Modifying the DataFrame in place is not supported by Dask. Defaults to False.

**compute**: bool Whether or not to trigger an immediate computation. Defaults to False.

### Examples

```python
>>> df2 = df.set_index('x')  # doctest: +SKIP
>>> df2 = df.set_index(d.x)  # doctest: +SKIP
>>> df2 = df.set_index(d.timestamp, sorted=True)  # doctest: +SKIP
```

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 is is separated

```python
>>> import pandas as pd
>>> divisions = pd.date_range('2000', '2010', freq='1D')
>>> df2 = df.set_index('timestamp', sorted=True, divisions=divisions)  # doctest: +SKIP
```

**shape**

Return a tuple representing the dimensionality of the DataFrame.

The number of rows is a Delayed result. The number of columns is a concrete integer.

### Examples

```python
>>> df.size  # doctest: +SKIP
(Delayed('int-07f06075-5ecc-4d77-817e-63c69a9188a8'), 2)
```

**shift** *(periods=1, freq=None, axis=0)*

Shift index by desired number of periods with an optional time freq.

This docstring was copied from pandas.core.frame.DataFrame.shift.

Some inconsistencies with the Dask version may exist.

When freq is not passed, shift the index without realigning the data. If freq is passed (in this case, the index must be date or datetime, or it will raise a NotImplementedError), the index will be increased using the periods and the freq.
Parameters

- **periods** [int] Number of periods to shift. Can be positive or negative.
- **freq** [DateOffset, tseries.offsets, timedelta, or str, optional] Offset to use from the tseries module or time rule (e.g. ‘EOM’). 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.
- **axis** [(0 or ‘index’, 1 or ‘columns’, None), default None] Shift direction.
- **fill_value** [object, optional (Not supported in Dask)] The scalar value to use for newly introduced missing values. The default depends on the dtype of self. For numeric data, np.nan is used. For datetime, timedelta, or period data, etc. NaT is used. For extension dtypes, self.dtype.na_value is used.

Changed in version 0.24.0.

Returns

**DataFrame** Copy of input object, shifted.

See also:

- **Index.shift** Shift values of Index.
- **DatetimeIndex.shift** Shift values of DatetimeIndex.
- **PeriodIndex.shift** Shift values of PeriodIndex.
- **tshift** Shift the time index, using the index’s frequency if available.

Examples

```python
>>> df = pd.DataFrame({'Col1': [10, 20, 15, 30, 45],  # doctest: +SKIP
... 'Col2': [13, 23, 18, 33, 48],
... 'Col3': [17, 27, 22, 37, 52]})

>>> df.shift(periods=3)  # doctest: +SKIP
    Col1  Col2  Col3
0     NaN  NaN  NaN
1     NaN  NaN  NaN
2     NaN  NaN  NaN
3    10.0  13.0  17.0
4    20.0  23.0  27.0

>>> df.shift(periods=1, axis='columns')  # doctest: +SKIP
    Col1  Col2  Col3
0     NaN  10.0  13.0
1     NaN  20.0  23.0
2     NaN  15.0  18.0
3     NaN  30.0  33.0
4     NaN  45.0  48.0

>>> df.shift(periods=3, fill_value=0)  # doctest: +SKIP
    Col1  Col2  Col3
0    0     0    0
1    0     0    0
2    0     0    0
```

(continues on next page)
size
Size of the Series or DataFrame as a Delayed object.

Examples

```python
>>> series.size  # doctest: +SKIP
dd.Scalar<size-ag..., dtype=int64>
```

squeeze *(axis=\text{None})*
Squeeze 1 dimensional axis objects into scalars.

This docstring was copied from pandas.core.frame.DataFrame.squeeze.

Some inconsistencies with the Dask version may exist.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or
a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don’t know if your object is a Series or DataFrame, but you do
know it has just a single column. In that case you can safely call `squeeze` to ensure you have a Series.

Parameters

- **axis** ([0 or ‘index’, 1 or ‘columns’, None], default None) A specific axis to squeeze. By
default, all length-1 axes are squeezed.

  New in version 0.20.0.

Returns

- **DataFrame, Series, or scalar** The projection after squeezing `axis` or all the axes.

See also:

- `Series.iloc` Integer-location based indexing for selecting scalars.
- `DataFrame.iloc` Integer-location based indexing for selecting Series.
- `Series.to_frame` Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```python
>>> primes = pd.Series([2, 3, 5, 7])  # doctest: +SKIP

Slicing might produce a Series with a single value:

```
>>> odd_primes = primes[primes % 2 == 1]  # doctest: +SKIP
1 3
2 5
3 7
dtype: int64

>>> odd_primes.squeeze()  # doctest: +SKIP
1 3
2 5
3 7
dtype: int64

Squeezing is even more effective when used with DataFrames.

```python
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])  # doctest: +SKIP
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```python
>>> df_a = df[['a']]  # doctest: +SKIP
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```python
>>> df_a.squeeze('columns')  # doctest: +SKIP
0 1
1 3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```python
>>> df_0a = df.loc[df.index < 1, ['a']]  # doctest: +SKIP
   a
0  1
```

Squeezing the rows produces a single scalar Series:

```python
>>> df_0a.squeeze('rows')  # doctest: +SKIP
   a
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```python
>>> df_0a.squeeze()  # doctest: +SKIP
1
```

```
std(axis=None, skipna=True, ddof=1, split_every=False, dtype=None, out=None)
Return sample standard deviation over requested axis.
```

This docstring was copied from pandas.core.frame.DataFrame.std.
Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series  
- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.  
- **numeric_only** [boolean, default None (Not supported in Dask)] 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)]

**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. With reverse version, rsub.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

**Parameters**

- **other** [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.  
- **axis** [{(0 or ‘index’, 1 or ‘columns’)}] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.  
- **level** [int or label] Broadcast across a level, matching Index values on the passed MultiIndex level.  
- **fill_value** [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

**Returns**

- **DataFrame** Result of the arithmetic operation.

**See also:**

- **DataFrame.add** Add DataFrames.
- **DataFrame.sub** Subtract DataFrames.
- **DataFrame.mul** Multiply DataFrames.
- **DataFrame.div** Divide DataFrames (float division).
- **DataFrame.truediv** Divide DataFrames (float division).
- **DataFrame.floordiv** Divide DataFrames (integer division).
**DataFrame.mod** Calculate modulo (remainder after division).

**DataFrame.pow** Calculate exponential power.

**Notes**

Mismatched indices will be unioned together.

**Examples**

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4], 'degrees': [360, 180, 360]}, index=['circle', 'triangle', 'rectangle'])

>>> df
  angles  degrees
circle    0      360
triangle  3      180
rectangle 4      360
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1
  angles  degrees
circle    1      361
triangle  4      181
rectangle 5      361
```

```python
>>> df.add(1)
  angles  degrees
circle    1      361
triangle  4      181
rectangle 5      361
```

Divide by constant with reverse version.

```python
>>> df.div(10)
  angles  degrees
circle   0.0    36.0
triangle 0.3    18.0
rectangle 0.4   36.0
```

```python
>>> df.rdiv(10)
  angles  degrees
circle    inf    0.027778
triangle 3.333333 0.055556
rectangle 2.500000 0.027778
```

Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]
  angles  degrees
circle   -1      358
triangle  2      178
rectangle  3      358
```
Multiply a DataFrame of different shape with operator version.

```python
other = pd.DataFrame({'angles': [0, 3, 4]}, index=['circle', 'triangle', 'rectangle'])
```

```python
df * other  # doctest: +SKIP
angles     degrees
circle     0.0 NaN
triangle   9.0 NaN
rectangle  16.0 NaN
```

Divide by a MultiIndex by level.

```python
df_multindex = pd.DataFrame({'angles': [0, 3, 4, 5, 6], 'degrees': [360, 180, 360, 360, 540, 720]}, index=['circle', 'triangle', 'rectangle', 'square', 'pentagon', 'hexagon'])
```

```python
df.div(df_multindex, level=1, fill_value=0)  # doctest: +SKIP
angles     degrees
A circle   NaN 1.0
```

(continues on next page)
sum (axis=None, skipna=True, split_every=False, dtype=None, out=None, min_count=None)

Return the sum of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.sum.

Some inconsistencies with the Dask version may exist.

This is equivalent to the method `numpy.sum`.

**Parameters**

- **axis** ([index (0), columns (1)]) Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** [bool, default None (Not supported in Dask)] 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 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

- ****kwargs Additional keyword arguments to be passed to the function.

**Returns**

- **sum** [Series or DataFrame (if level specified)]

See also:

- `Series.sum` Return the sum.
- `Series.min` Return the minimum.
- `Series.max` Return the maximum.
- `Series.idxmin` Return the index of the minimum.
- `Series.idxmax` Return the index of the maximum.
- `DataFrame.min` Return the sum over the requested axis.
- `DataFrame.min` Return the minimum over the requested axis.
- `DataFrame.max` Return the maximum over the requested axis.
- `DataFrame.idxmin` Return the index of the minimum over the requested axis.
- `DataFrame.idxmax` Return the index of the maximum over the requested axis.
Examples

```python
>>> idx = pd.MultiIndex.from_arrays([  # doctest: +SKIP
  ... ['warm', 'warm', 'cold', 'cold'],
  ... ['dog', 'falcon', 'fish', 'spider'],
  ... names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)  # doctest: +SKIP
>>> s
blooded animal
  warm  dog 4
  cold  fish 0
  cold  spider 8
Name: legs, dtype: int64
```

```python
>>> s.sum()  # doctest: +SKIP
14
```

Sum using level names, as well as indices.

```python
>>> s.sum(level='blooded')  # doctest: +SKIP
blooded
  warm 6
  cold 8
Name: legs, dtype: int64
```

```python
>>> s.sum(level=0)  # doctest: +SKIP
blooded
  warm 6
  cold 8
Name: legs, dtype: int64
```

By default, the sum of an empty or all-NA Series is 0.

```python
>>> pd.Series([]).sum()  # min_count=0 is the default  # doctest: +SKIP
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`.

```python
>>> pd.Series([]).sum(min_count=1)  # doctest: +SKIP
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).sum()  # doctest: +SKIP
0.0
```

```python
>>> pd.Series([np.nan]).sum(min_count=1)  # doctest: +SKIP
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

```python
>>> bag = df.to_bag()  # doctest: +SKIP
```

**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:

```python
>>> 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.

```python
>>> from datetime import date, timedelta
>>> def name(i):
...    return str(date(2015, 1, 1) + i * timedelta(days=1))
```

```python
>>> name(0)
'2015-01-01'
>>> name(15)
'2015-01-16'
```

```python
>>> df.to_csv('/path/to/data/export-*.csv', name_function=name)  # doctest: +SKIP
```

```
/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:

```python
>>> 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

 header_first_partition_only [boolean, default False] If set, only write the header row in the first output file

 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_dask_array** *(lengths=None)*

Convert a dask DataFrame to a dask array.

**Parameters**

- **lengths** [bool or Sequence of ints, optional] How to determine the chunks sizes for the output array. By default, the output array will have unknown chunk lengths along the first axis, which can cause some later operations to fail.
  - True: immediately compute the length of each partition
  - Sequence: a sequence of integers to use for the chunk sizes on the first axis. These values are not validated for correctness, beyond ensuring that the number of items matches the number of partitions.

**to_delayed** *(optimize_graph=True)*

Convert into a list of dask.delayed objects, one per partition.

**Parameters**

- **optimize_graph** [bool, optional] If True [default], the graph is optimized before converting into dask.delayed objects.

See also:

dask.dataframe.from_delayed

**Examples**

```python
>>> partitions = df.to_delayed()  # doctest: +SKIP
```

**to_hdf** *(path_or_buf, key, mode='a', append=False, **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 asterix * within the filename or datapath, and an optional name_function. The asterix 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.
scheduler  [string] The scheduler to use, like “threads” or “processes”

**other:**  See pandas.to_hdf for more information

Returns

filenames  [list] Returned if compute is True. List of file names that each partition is saved to.

delayed  [dask.Delayed] Returned if compute is False. Delayed object to execute to_hdf when computed.

See also:
read_hdf, to_parquet

Examples

Save Data to a single file

```python
>>> df.to_hdf('output.hdf', '/data')  # doctest: +SKIP
```

Save data to multiple datapaths within the same file:

```python
>>> df.to_hdf('output.hdf', '/data-*')  # doctest: +SKIP
```

Save data to multiple files:

```python
>>> df.to_hdf('output-*.hdf', '/data')  # doctest: +SKIP
```

Save data to multiple files, using the multiprocessing scheduler:

```python
>>> df.to_hdf('output-*.hdf', '/data', scheduler='processes')  # doctest: +SKIP
```


```python
>>> 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)  # doctest: +SKIP
```

to_html (max_rows=5)
Render a DataFrame as an HTML table.

Parameters

buf  [StringIO-like, optional (Not supported in Dask)] Buffer to write to.

columns  [sequence, optional, default None (Not supported in Dask)] The subset of columns to write. Writes all columns by default.

col_space  [int, optional (Not supported in Dask)] The minimum width of each column.

header  [bool, optional (Not supported in Dask)] Whether to print column labels, default True.
index [bool, optional, default True (Not supported in Dask)] Whether to print index (row) labels.

na_rep [str, optional, default ‘NaN’ (Not supported in Dask)] String representation of NAN to use.

formatters [list or dict of one-param. functions, optional (Not supported in Dask)] Formatter functions to apply to columns’ elements by position or name. 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, default None (Not supported in Dask)] Formatter function to apply to columns’ elements if they are floats. The result of this function must be a unicode string.

sparsify [bool, optional, default True (Not supported in Dask)] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names [bool, optional, default True (Not supported in Dask)] Prints the names of the indexes.

justify [str, default None (Not supported in Dask)] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

max_rows [int, optional] Maximum number of rows to display in the console.

max_cols [int, optional (Not supported in Dask)] Maximum number of columns to display in the console.

show_dimensions [bool, default False (Not supported in Dask)] Display DataFrame dimensions (number of rows by number of columns).

decimal [str, default ‘.’ (Not supported in Dask)] Character recognized as decimal separator, e.g. ‘,’ in Europe.

bold_rows [bool, default True (Not supported in Dask)] Make the row labels bold in the output.
classes [str or list or tuple, default None (Not supported in Dask)] CSS class(es) to apply to the resulting html table.

escape [bool, default True (Not supported in Dask)] Convert the characters <, >, and & to HTML-safe sequences.

notebook [{True, False}, default False (Not supported in Dask)] Whether the generated HTML is for IPython Notebook.

border [int (Not supported in Dask)] A border=border attribute is included in the opening <table> tag. Default pd.options.html.border.

New in version 0.19.0.

table_id [str, optional (Not supported in Dask)] A css id is included in the opening <table> tag if specified.

New in version 0.23.0.

render_links [bool, default False (Not supported in Dask)] Convert URLs to HTML links.

New in version 0.24.0.

Returns

str (or unicode, depending on data and options) String representation of the dataframe.

See also:

to_string Convert DataFrame to a string.

to_json (filename, *args, **kwargs)
See dd.to_json docstring for more information

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.

ingine [{‘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

```python
>>> df = dd.read_csv(...)  # doctest: +SKIP
>>> dd.to_parquet(df, '/path/to/output/', compression='snappy')  # doctest: +SKIP
```

to_records (**index=False**)  
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

```python
>>> df.to_records()  # doctest: +SKIP
dask.array<shape=(nan,), dtype=(numpy.record, [('ind', '<f8'), ('x', 'O'), ('y', '<i8')]), chunksize=(nan,)>
```

to_string (**max_rows=5**)  
Render a DataFrame to a console-friendly tabular output.

Parameters

- **buf** [StringIO-like, optional (Not supported in Dask)] Buffer to write to.
- **columns** [sequence, optional, default None (Not supported in Dask)] The subset of columns to write. Writes all columns by default.
- **col_space** [int, optional (Not supported in Dask)] The minimum width of each column.
- **header** [bool, optional (Not supported in Dask)] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.
index [bool, optional, default True (Not supported in Dask)] Whether to print index (row) labels.

na_rep [str, optional, default ‘NaN’ (Not supported in Dask)] String representation of NaN to use.

formatters [list or dict of one-param. functions, optional (Not supported in Dask)] Formatter functions to apply to columns’ elements by position or name. 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, default None (Not supported in Dask)] Formatter function to apply to columns’ elements if they are floats. The result of this function must be a unicode string.

sparsify [bool, optional, default True (Not supported in Dask)] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

index_names [bool, optional, default True (Not supported in Dask)] Prints the names of the indexes.

justify [str, default None (Not supported in Dask)] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), ‘right’ out of the box. Valid values are

left
right
center
justify
justify-all
start
end
inherit
match-parent
initial
unset.

max_rows [int, optional] Maximum number of rows to display in the console.

max_cols [int, optional (Not supported in Dask)] Maximum number of columns to display in the console.

show_dimensions [bool, default False (Not supported in Dask)] Display DataFrame dimensions (number of rows by number of columns).

decimal [str, default ‘.’ (Not supported in Dask)] Character recognized as decimal separator, e.g. ‘,’ in Europe.

New in version 0.18.0.

line_width [int, optional (Not supported in Dask)] Width to wrap a line in characters.

Returns
str (or unicode, depending on data and options) String representation of the dataframe.

See also:

to_html Convert DataFrame to HTML.

Examples

```python
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}  # doctest: +SKIP
>>> df = pd.DataFrame(d)  # doctest: +SKIP
>>> print(df.to_string())  # doctest: +SKIP
     col1  col2
0      1    4
1      2    5
2      3    6
```

to_timestamp (freq=None, how='start', axis=0) Cast to DatetimeIndex of timestamps, at beginning of period.

This docstring was copied from pandas.core.frame.DataFrame.to_timestamp. Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] If false then underlying input data is not copied

Returns

df [DataFrame with DatetimeIndex]

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. With reverse version, rtruediv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- `other` [scalar, sequence, Series, or DataFrame] Any single or multiple element data structure, or list-like object.
- `axis` [0 or ‘index’, 1 or ‘columns’] Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- `level` [int or label] Broadcast across a level, matching Index values on the passed Multi-Index level.
- `fill_value` [float or None, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.
Returns

DataFrame  Result of the arithmetic operation.

See also:

DataFrame.add  Add DataFrames.
DataFrame.sub  Subtract DataFrames.
DataFrame.mul  Multiply DataFrames.
DataFrame.div  Divide DataFrames (float division).
DataFrame.truediv  Divide DataFrames (float division).
DataFrame.floordiv  Divide DataFrames (integer division).
DataFrame.mod  Calculate modulo (remainder after division).
DataFrame.pow  Calculate exponential power.

Notes

Mismatched indices will be unioned together.

Examples

```python
>>> df = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
...                   'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
```

Add a scalar with operator version which return the same results.

```python
>>> df + 1  # doctest: +SKIP
angles  degrees
circle  1  361
triangle  4  181
rectangle  5  361
```

```python
>>> df.add(1)  # doctest: +SKIP
angles  degrees
circle  1  361
triangle  4  181
rectangle  5  361
```

Divide by constant with reverse version.

```python
>>> df.div(10)  # doctest: +SKIP
angles  degrees
circle  0.0  36.0
triangle  0.3  18.0
rectangle  0.4  36.0
```
Subtract a list and Series by axis with operator version.

```python
>>> df - [1, 2]  # doctest: +SKIP
   angles  degrees
circle    2      358
triangle  2      178
rectangle 3      358
```

```python
>>> df.sub([1, 2], axis='columns')  # doctest: +SKIP
   angles  degrees
circle    1      358
triangle  2      178
rectangle 3      358
```

```python
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']), # doctest: +SKIP
   axis='index')
   angles  degrees
circle    1      359
triangle  2      179
rectangle 3      359
```

Multiply a DataFrame of different shape with operator version.

```python
>>> other = pd.DataFrame({'angles': [0, 3, 4],  # doctest: +SKIP
   ... index=['circle', 'triangle', 'rectangle']})
>>> df * other  # doctest: +SKIP
   angles  degrees
circle    0      NaN
triangle  9      NaN
rectangle 16     NaN
```

```python
>>> df.mul(other, fill_value=0)  # doctest: +SKIP
   angles  degrees
circle    0      0.0
triangle  9      0.0
rectangle 16     0.0
```

Divide by a MultiIndex by level.

```python
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],  # doctest: +SKIP
   ... 'degrees': [360, 180, 360, 360, 540, 720],
   ... index=['A', 'A', 'A', 'B', 'B', 'B'],
   ...})
```

(continues on next page)
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

Return unbiased variance over requested axis.

This docstring was copied from pandas.core.frame.DataFrame.var.

Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddf [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddf, where N represents the number of elements.

numeric_only [boolean, default None (Not supported in Dask)] 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)]

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.

color: [None, 'order'], optional Options to color nodes. Provide cmap= keyword for additional colormap

**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:

https://docs.dask.org/en/latest/optimize.html

Examples

```python
>>> x.visualize(filename='dask.pdf')  # doctest: +SKIP
>>> x.visualize(filename='dask.pdf', color='order')  # doctest: +SKIP
```

where (cond, other=nan)

Replace values where the condition is False.

This docstring was copied from pandas.core.frame.DataFrame.where. Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] Whether to perform the operation in place on the data.
The where method is an application of the if-then idiom. For each element in the calling DataFrame, if \texttt{cond} is \texttt{True} the element is used; otherwise the corresponding element from the DataFrame \texttt{other} is used.

The signature for \texttt{DataFrame.where()} differs from \texttt{numpy.where()}. Roughly \texttt{df1.where(m, df2)} is equivalent to \texttt{np.where(m, df1, df2)}.

For further details and examples see the \texttt{where} documentation in indexing.

**Examples**

```python
code
>>> s = pd.Series(range(5))  # doctest: +SKIP
>>> s where(s > 0)  # doctest: +SKIP
0    NaN
1      1.0
2      2.0
3      3.0
4      4.0
dtype: float64
```

```python
code
>>> s.mask(s > 0)  # doctest: +SKIP
0      0.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```
>>> s.where(s > 1, 10)  # doctest: +SKIP
0   10
1    2
2    3
3    4
4    6
dtype: int64

>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])  # doctest: +SKIP

>>> m = df % 3 == 0  # doctest: +SKIP

>>> df.where(m, -df)  # doctest: +SKIP
   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)  # doctest: +SKIP
   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 a Series/DataFrame with absolute numeric value of each element.
This docstring was copied from pandas.core.frame.DataFrame.abs. Some inconsistencies with the Dask version may exist.
This function only applies to elements that are all numeric.

**Returns**

*abs*  Series/DataFrame containing the absolute value of each element.

**See also:**

*numpy.absolute* Calculate the absolute value element-wise.

**Notes**

For complex inputs, \(1.2 + 1j\), the absolute value is \(\sqrt{a^2 + b^2}\).

**Examples**

Absolute numeric values in a Series.

```python
>>> s = pd.Series([-1.10, 2, -3.33, 4])  # doctest: +SKIP
>>> s.abs()  # doctest: +SKIP
0  1.10
1  2.00
2  3.33
3  4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```python
>>> s = pd.Series([1.2 + 1j])  # doctest: +SKIP
>>> s.abs()  # doctest: +SKIP
0  1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```python
>>> s = pd.Series([pd.Timedelta('1 days')])  # doctest: +SKIP
>>> s.abs()  # doctest: +SKIP
0  1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```python
>>> df = pd.DataFrame({  # doctest: +SKIP
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df  # doctest: +SKIP
    a  b    c
0  4  10  100
1  5  20   50
2  6  30 - 30
3  7  40  -50
```

(continues on next page)
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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is 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

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a  # doctest: +SKIP
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b  # doctest: +SKIP
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```
align (other, join='outer', axis=None, fill_value=None)
Align two objects on their axes with the specified join method for each axis Index.

This docstring was copied from pandas.core.series.Series.align.

Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] Broadcast across a level, matching Index values on the passed MultiIndex level
copy [boolean, default True (Not supported in Dask)] 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 [{‘backfill’, ‘bfill’, ‘pad’, ‘ffill’}, default None (Not supported in Dask)] 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
limit [int, default None (Not supported in Dask)] 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.
fill_axis [{0 or ‘index’}, default 0 (Not supported in Dask)] Filling axis, method and limit
broadcast_axis [{0 or ‘index’}, default None (Not supported in Dask)] Broadcast values along this axis, if aligning two objects of different dimensions

Returns

(left, right) [(Series, type of other)] Aligned objects

all (axis=None, skipna=True, split_every=False, out=None)
Return whether all elements are True, potentially over an axis.

This docstring was copied from pandas.core.frame.DataFrame.all.

Some inconsistencies with the Dask version may exist.

Returns True unless there at least one element within a series or along a DataFrame axis that is False or equivalent (e.g. zero or empty).

Parameters

axis [{0 or ‘index’, 1 or ‘columns’}, default 0] Indicate which axis or axes should be reduced.

• 0 / ‘index’ : reduce the index, return a Series whose index is the original column labels.
• 1 / ‘columns’ : reduce the columns, return a Series whose index is the original index.
• None : reduce all axes, return a scalar.

**bool_only** [bool, default None (Not supported in Dask)] Include only boolean columns.
If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**skipna** [bool, default True] Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

**level** [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

**kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

Series or DataFrame If level is specified, then, DataFrame is returned; otherwise, Series is returned.

See also:

**Series.all** Return True if all elements are True.

**DataFrame.any** Return True if one (or more) elements are True.

Examples

Series

```python
>>> pd.Series([True, True]).all()  # doctest: +SKIP
True
>>> pd.Series([True, False]).all()  # doctest: +SKIP
False
>>> pd.Series([]).all()  # doctest: +SKIP
True
>>> pd.Series([np.nan]).all()  # doctest: +SKIP
True
>>> pd.Series([np.nan]).all(skipna=False)  # doctest: +SKIP
True
```

DataFrames

Create a dataframe from a dictionary.

```python
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})  # doctest: +SKIP
>>> df  # doctest: +SKIP
coll  col2
0 True  True
1 True  False

Default behaviour checks if column-wise values all return True.

```python
>>> df.all()  # doctest: +SKIP
coll  True
col2  False
dtype: bool
```
Specify `axis='columns'` to check if row-wise values all return True.

```python
>>> df.all(axis='columns')  # doctest: +SKIP
0  True
1  False
dtype: bool
```

Or `axis=None` for whether every value is True.

```python
>>> df.all(axis=None)  # doctest: +SKIP
False
```

`any` *(axis=None, skipna=True, split_every=False, out=None)*

Return whether any element is True, potentially over an axis.

This docstring was copied from pandas.core.frame.DataFrame.any.

Some inconsistencies with the Dask version may exist.

Returns False unless there at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

**Parameters**

- **axis** *(0 or ‘index’, 1 or ‘columns’, None), default 0* Indicate which axis or axes should be reduced.
  
  0 / ‘index’ : reduce the index, return a Series whose index is the original column labels.
  
  1 / ‘columns’ : reduce the columns, return a Series whose index is the original index.
  
  None : reduce all axes, return a scalar.

- **bool_only** *(bool, default None (Not supported in Dask))* Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

- **skipna** *(bool, default True)* Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.

- **level** *(int or level name, default None (Not supported in Dask))* If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

- ****kwags** *(any, default None)* Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **Series or DataFrame** If level is specified, then, DataFrame is returned; otherwise, Series is returned.

**See also:**

- **numpy.any** Numpy version of this method.
- **Series.any** Return whether any element is True.
- **Series.all** Return whether all elements are True.
- **DataFrame.any** Return whether any element is True over requested axis.
**DataFrame.all**  Return whether all elements are True over requested axis.

**Examples**

**Series**

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()  # doctest: +SKIP
False
>>> pd.Series([True, False]).any()  # doctest: +SKIP
True
>>> pd.Series([]).any()  # doctest: +SKIP
False
>>> pd.Series([np.nan]).any()  # doctest: +SKIP
False
>>> pd.Series([np.nan]).any(skipna=False)  # doctest: +SKIP
True
```

**DataFrame**

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0, 2], 'C': [0, 0]})  # doctest: +SKIP
>>> df
  A  B  C
0 1  0  0
1 2  2  0
```

```
>>> df.any()  # doctest: +SKIP
A  True
B  True
C  False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({'A': [True, False], 'B': [1, 2]})  # doctest: +SKIP
>>> df
  A  B
0  True 1
1  False 2
```

```
>>> df.any(axis='columns')  # doctest: +SKIP
0  True
1  True
dtype: bool
```

```
>>> df = pd.DataFrame({'A': [True, False], 'B': [1, 0]})  # doctest: +SKIP
>>> df
  A  B
0  True 1
1  False 0
```

```
4.9. DataFrame 577
```
Aggregating over the entire DataFrame with `axis=None`.

```python
>>> df.any(axis='columns')  # doctest: +SKIP
0    True
1    False
dtype: bool
```

`any` for an empty DataFrame is an empty Series.

```python
>>> pd.DataFrame([]).any()  # doctest: +SKIP
Series([], dtype: bool)
```

### `append` (other, interleave_partitions=False)

Concatenate two or more Series.

This docstring was copied from pandas.core.series.Series.append.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **to_append** [Series or list/tuple of Series (Not supported in Dask)]
- **ignore_index** [boolean, default False (Not supported in Dask)] If True, do not use the index labels.
  
  New in version 0.19.0.
- **verify_integrity** [boolean, default False (Not supported in Dask)] If True, raise Exception on creating index with duplicates

**Returns**

- **appended** [Series]

**See also:**

- **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**

```python
>>> s1 = pd.Series([1, 2, 3])  # doctest: +SKIP
>>> s2 = pd.Series([4, 5, 6])  # doctest: +SKIP
>>> s3 = pd.Series([4, 5, 6], index=[3, 4, 5])  # doctest: +SKIP
>>> s1.append(s2)  # doctest: +SKIP
0  1
1  2
2  3
0  4
1  5
```
With `ignore_index` set to True:

```python
>>> s1.append(s2, ignore_index=True)  # doctest: +SKIP
0 1
1 2
2 3
3 4
4 5
5 6
dtype: int64
```

With `verify_integrity` set to True:

```python
>>> s1.append(s2, verify_integrity=True)  # doctest: +SKIP
Traceback (most recent call last):
... 
ValueError: Indexes have overlapping values: [0, 1, 2]
```

**apply** *(func, convert_dtype=True, meta='__no_default__', args=(), **kwds)*

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

```python
>>> 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`:

```python
>>> 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:

```python
>>> 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:

```python
>>> res = ds.apply(lambda x: x + 1, meta=ds)
```

`astype (dtype)`

Cast a pandas object to a specified `dtype` `dtype`.

This docstring was copied from pandas.core.frame.DataFrame.astype. Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] 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' (Not supported in Dask)] 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.

- `kwargs` [keyword arguments to pass on to the constructor]

Returns

- `casted` [same type as caller]

See also:
to_datetime  Convert argument to datetime.

to_timedelta  Convert argument to timedelta.

to_numeric  Convert argument to a numeric type.

numpy.ndarray.astype  Cast a numpy array to a specified type.

Examples

```python
>>> ser = pd.Series([1, 2], dtype='int32')  # doctest: +SKIP
>>> ser  # doctest: +SKIP
0    1
1    2
dtype: int32

>>> ser.astype('int64')  # doctest: +SKIP
0    1
1    2
dtype: int64

Convert to categorical type:

```python
>>> ser.astype('category')  # doctest: +SKIP
0 1
1 2
dtype: category
Categories (2, int64): [1, 2]

Convert to ordered categorical type with custom ordering:

```python
>>> cat_dtype = pd.api.types.CategoricalDtype(...
...    categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)  # doctest: +SKIP
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:

```python
>>> s1 = pd.Series([1,2])  # doctest: +SKIP
>>> s2 = s1.astype('int64', copy=False)  # doctest: +SKIP
>>> s2[0] = 10  # doctest: +SKIP
>>> s1  # note that s1[0] has changed too  # doctest: +SKIP
0    10
1    2
dtype: int64
```

autocorr (lag=1, split_every=False)

Compute the lag-N autocorrelation.

This docstring was copied from pandas.core.series.Series.autocorr.

Some inconsistencies with the Dask version may exist.

This method computes the Pearson correlation between the Series and its shifted self.

Parameters

- lag  [int, default 1] Number of lags to apply before performing autocorrelation.
Returns

float  The Pearson correlation between self and self.shift(lag).

See also:

Series.corr  Compute the correlation between two Series.
Series.shift  Shift index by desired number of periods.
DataFrame.corr  Compute pairwise correlation of columns.
DataFrame.corrwith  Compute pairwise correlation between rows or columns of two DataFrame objects.

Notes

If the Pearson correlation is not well defined return ‘NaN’.

Examples

```python
>>> s = pd.Series([0.25, 0.5, 0.2, -0.05])  # doctest: +SKIP
>>> s.autocorr()  # doctest: +ELLIPSIS, +SKIP
0.10355...
>>> s.autocorr(lag=2)  # doctest: +ELLIPSIS, +SKIP
-0.99999...
```

If the Pearson correlation is not well defined, then ‘NaN’ is returned.

```python
>>> s = pd.Series([1, 0, 0, 0])  # doctest: +SKIP
>>> s.autocorr()  # doctest: +SKIP
nan
```

between (left, right, inclusive=True)

Return boolean Series equivalent to left <= series <= right.

This docstring was copied from pandas.core.series.Series.between.

Some inconsistencies with the Dask version may exist.

This function returns a boolean vector containing True wherever the corresponding Series element is between the boundary values left and right. NA values are treated as False.

Parameters

left  [scalar] Left boundary.
right  [scalar] Right boundary.
inclusive  [bool, default True] Include boundaries.

Returns

Series  Each element will be a boolean.

See also:

Series.gt  Greater than of series and other.
Series.lt  Less than of series and other.
Notes

This function is equivalent to `(left <= ser) & (ser <= right)`

Examples

```python
>>> s = pd.Series([2, 0, 4, 8, np.nan])  # doctest: +SKIP
Boundary values are included by default:

```s.between(1, 4)  # doctest: +SKIP
0   True
1   False
2   True
3   False
4   False
dtype: bool
```

With `inclusive` set to `False` boundary values are excluded:

```python
>>> s.between(1, 4, inclusive=False)  # doctest: +SKIP
0   True
1   False
2   False
3   False
4   False
dtype: bool
```

`left` and `right` can be any scalar value:

```python
>>> s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])  # doctest: +SKIP
>>> s.between('Anna', 'Daniel')  # doctest: +SKIP
0   False
1   True
2   True
3   False
dtype: bool
```

`bfill` *(axis=None, limit=None)*

Synonym for `DataFrame.fillna()` with method='bfill'.

`clear_divisions()`

Forget division information

`clip` *(lower=None, upper=None, out=None)*

Trim values at input threshold(s).

This docstring was copied from pandas.core.series.Series.clip.

Some inconsistencies with the Dask version may exist.

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

**Parameters**

- `lower` [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.
**upper** [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

**axis** [int or string axis name, optional (Not supported in Dask)] Align object with lower and upper along the given axis.

**inplace** [boolean, default False (Not supported in Dask)] Whether to perform the operation in place on the data.

New in version 0.21.0.

*args, **kwargs Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns**

*Series or DataFrame* Same type as calling object with the values outside the clip boundaries replaced.

**Examples**

```python
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}  # doctest: +SKIP
>>> df = pd.DataFrame(data)  # doctest: +SKIP
>>> df  # doctest: +SKIP
   col_0  col_1
0     9    -2
1    -3    -7
2     0     6
3    -1     8
4     5    -5
```

Clips per column using lower and upper thresholds:

```python
>>> df.clip(-4, 6)  # doctest: +SKIP
   col_0  col_1
0     6    -2
1    -3    -4
2     0     6
3    -1     6
4     5    -4
```

Clips using specific lower and upper thresholds per column element:

```python
>>> t = pd.Series([2, -4, -1, 6, 3])  # doctest: +SKIP
>>> t  # doctest: +SKIP
0  2
1 -4
2 -1
3  6
4  3
dtype: int64
```

```python
>>> df.clip(t, t + 4, axis=0)  # doctest: +SKIP
   col_0  col_1
0     6     2
1    -3    -4
2     0     3
```

(continues on next page)
`clip_lower(threshold)`

Trim values below a given threshold.

This docstring was copied from pandas.core.series.Series.clip_lower.

Some inconsistencies with the Dask version may exist.

Deprecated since version 0.24.0: Use `clip(lower=threshold)` instead.

Elements below the `threshold` will be changed to match the `threshold` value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.

**Parameters**

- **threshold** [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.
  - float: every value is compared to `threshold`.
  - array-like: The shape of `threshold` should match the object it’s compared to. When `self` is a Series, `threshold` should be the length. When `self` is a DataFrame, `threshold` should be 2-D and the same shape as `self` for `axis=None`, or 1-D and the same length as the axis being compared.

- **axis** [{0 or 'index', 1 or 'columns'}, default 0 (Not supported in Dask)] Align `self` with `threshold` along the given axis.

- **inplace** [boolean, default False (Not supported in Dask)] Whether to perform the operation in place on the data.

**Returns**

- **Series or DataFrame** Original data with values trimmed.

**See also:**

- `Series.clip` General purpose method to trim Series values to given threshold(s).
- `DataFrame.clip` General purpose method to trim DataFrame values to given threshold(s).

**Examples**

Series single threshold clipping:

```python
>>> s = pd.Series([5, 6, 7, 8, 9])  # doctest: +SKIP
>>> s.clip(lower=8)  # doctest: +SKIP
0 8
1 8
2 8
3 8
4 9
dtype: int64
```

Series clipping element-wise using an array of thresholds. `threshold` should be the same length as the Series.
elemwise_thresholds = [4, 8, 7, 2, 5]  # doctest: +SKIP
s.clip(lower=elemwise_thresholds)  # doctest: +SKIP
0  5  
1  8  
2  7  
3  8  
4  9  
dtype: int64

dataFrames can be compared to a scalar.

def = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})  # doctest: +SKIP
def  # doctest: +SKIP
   A  B
0  1  2
1  3  4
2  5  6

def.clip(lower=3)  # doctest: +SKIP
   A  B
0  3  3
1  3  4
2  5  6

Or to an array of values. By default, threshold should be the same shape as the DataFrame.

def.clip(lower=np.array([[3, 4], [2, 2], [6, 2]]))  # doctest: +SKIP
   A  B
0  3  4
1  3  4
2  6  6

Control how threshold is broadcast with axis. In this case threshold should be the same length as the axis specified by axis.

def.clip(lower=[3, 3, 5], axis='index')  # doctest: +SKIP
   A  B
0  3  3
1  3  4
2  5  6

def.clip(lower=[4, 5], axis='columns')  # doctest: +SKIP
   A  B
0  4  5
1  4  5
2  5  6

clip_upper(threshold)
Trim values above a given threshold.

This docstring was copied from pandas.core.series.Series.clip_upper.

Some inconsistencies with the Dask version may exist.

Deprecated since version 0.24.0: Use clip(upper=threshold) instead.

Elements above the threshold will be changed to match the threshold value(s). Threshold can be a single value or an array, in the latter case it performs the truncation element-wise.
Parameters

**threshold** [numeric or array-like] Maximum value allowed. All values above threshold will be set to this value.

- float: every value is compared to **threshold**.
- array-like: The shape of **threshold** should match the object it’s compared to. When **self** is a Series, **threshold** should be the length. When **self** is a DataFrame, **threshold** should 2-D and the same shape as **self** for **axis=None**, or 1-D and the same length as the axis being compared.

**axis** [{0 or ‘index’, 1 or ‘columns’}, default 0 (Not supported in Dask)] Align object with **threshold** along the given axis.

**inplace** [boolean, default False (Not supported in Dask)] Whether to perform the operation in place on the data.

    New in version 0.21.0.

Returns

**Series or DataFrame** Original data with values trimmed.

See also:

**Series.clip** General purpose method to trim Series values to given threshold(s).

**DataFrame.clip** General purpose method to trim DataFrame values to given threshold(s).

Examples

```python
>>> s = pd.Series([1, 2, 3, 4, 5])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0 1
1 2
2 3
3 4
4 5
dtype: int64

>>> s.clip(upper=3)  # doctest: +SKIP
0 1
1 2
2 3
3 3
4 3
dtype: int64

>>> elemwise_thresholds = [5, 4, 3, 2, 1]  # doctest: +SKIP
>>> elemwise_thresholds  # doctest: +SKIP
[5, 4, 3, 2, 1]

>>> s.clip(upper=elemwise_thresholds)  # doctest: +SKIP
0 1
1 2
2 3
3 2
```

(continues on next page)
**combine** *(other, func, fill_value=None)*

Combine the Series with a Series or scalar according to `func`.

This docstring was copied from pandas.core.series.Series.combine.

Some inconsistencies with the Dask version may exist.

Combine the Series and *other* using `func` to perform elementwise selection for combined Series. `fill_value` is assumed when value is missing at some index from one of the two objects being combined.

**Parameters**

- **other** [Series or scalar] The value(s) to be combined with the `Series`.
- **func** [function] Function that takes two scalars as inputs and returns an element.
- **fill_value** [scalar, optional] The value to assume when an index is missing from one Series or the other. The default specifies to use the appropriate NaN value for the underlying dtype of the Series.

**Returns**

- **Series** The result of combining the Series with the other object.

**See also:**

- `Series.combine_first` Combine Series values, choosing the calling Series’ values first.

**Examples**

Consider 2 Datasets `s1` and `s2` containing highest clocked speeds of different birds.

```python
>>> s1 = pd.Series({'falcon': 330.0, 'eagle': 160.0})  # doctest: +SKIP
>>> s1
falcon    330.0
eagle     160.0
dtype: float64

>>> s2 = pd.Series({'falcon': 345.0, 'eagle': 200.0, 'duck': 30.0})  # doctest: +SKIP
>>> s2
falcon    345.0
eagle     200.0
duck      30.0
dtype: float64
```

Now, to combine the two datasets and view the highest speeds of the birds across the two datasets

```python
>>> s1.combine(s2, max)  # doctest: +SKIP
duck    NaN
eagle   200.0
falcon  345.0
dtype: float64
```

In the previous example, the resulting value for duck is missing, because the maximum of a NaN and a float is a NaN. So, in the example, we set `fill_value=0`, so the maximum value returned will be the value from some dataset.
```python
>>> s1.combine(s2, max, fill_value=0)  # doctest: +SKIP
duck  30.0
eagle 200.0
falcon 345.0
dtype: float64
```

**combine_first** (other)

Combine Series values, choosing the calling Series’s values first.

This docstring was copied from pandas.core.series.Series.combine_first.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **other** [Series] The value(s) to be combined with the Series.

**Returns**

- **Series** The result of combining the Series with the other object.

**See also:**

- **Series.combine** Perform elementwise operation on two Series using a given function.

**Notes**

Result index will be the union of the two indexes.

**Examples**

```python
>>> s1 = pd.Series([1, np.nan])  # doctest: +SKIP
>>> s2 = pd.Series([3, 4])  # doctest: +SKIP
>>> s1.combine_first(s2)  # doctest: +SKIP
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**

- **scheduler** [string, optional] Which scheduler to use like “threads”, “synchronous” or “processes”. 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 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.

This docstring was copied from pandas.core.series.Series.corr.

Some inconsistencies with the Dask version may exist.

Parameters

other [Series]
method [{‘pearson’, ‘kendall’, ‘spearman’} or callable]
• pearson : standard correlation coefficient
• kendall : Kendall Tau correlation coefficient
• spearman : Spearman rank correlation
• callable: callable with input two 1d ndarray and returning a float .. version-added:: 0.24.0

min_periods [int, optional] Minimum number of observations needed to have a valid result

Returns
correlation [float]

Examples

```python
>>> histogram_intersection = lambda a, b: np.minimum(a, b  # doctest: +SKIP
... ).sum().round(decimals=1)
>>> s1 = pd.Series([.2, .0, .6, .2])  # doctest: +SKIP
>>> s2 = pd.Series([.3, .6, .0, .1])  # doctest: +SKIP
>>> s1.corr(s2, method=histogram_intersection)  # doctest: +SKIP
0.3
```

count (split_every=False)
Return number of non-NA/null observations in the Series.

This docstring was copied from pandas.core.series.Series.count.

Some inconsistencies with the Dask version may exist.

Parameters

level [int or level name, default None (Not supported in Dask)] 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)]

cov (other, min_periods=None, split_every=False)
Compute covariance with Series, excluding missing values.

This docstring was copied from pandas.core.series.Series.cov.
Some inconsistencies with the Dask version may exist.

**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).

```python
cummax (axis=None, skipna=True, out=None)
```

Return cumulative maximum over a DataFrame or Series axis.

This docstring was copied from pandas.core.frame.DataFrame.cummax.
Some inconsistencies with the Dask version may exist.
Returns a DataFrame or Series of the same size containing the cumulative maximum.

**Parameters**

- `axis` [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.
- `skipna` [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- `*args, **kwargs` Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- `cummax` [Series or DataFrame]

See also:

- `core.window.Expanding.max` Similar functionality but ignores NaN values.
- `DataFrame.max` Return the maximum over DataFrame axis.
- `DataFrame.cummax` Return cumulative maximum over DataFrame axis.
- `DataFrame.cummin` Return cumulative minimum over DataFrame axis.
- `DataFrame.cumsum` Return cumulative sum over DataFrame axis.
- `DataFrame.cumprod` Return cumulative product over DataFrame axis.

**Examples**

**Series**

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0  2.0
1  NaN
2  5.0
3 -1.0
4  0.0
dtype: float64
```
By default, NA values are ignored.

```python
>>> s.cummax()  # doctest: +SKIP
0  2.0
1  NaN
2  5.0
3  5.0
4  5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```python
>>> s.cummax(skipna=False)  # doctest: +SKIP
0  2.0
1  NaN
2  NaN
3  NaN
4  NaN
dtype: float64
```

**DataFrame**

```python
>>> df = pd.DataFrame([[2.0, 1.0],
...                     [3.0, np.nan],
...                     [1.0, 0.0]],
...                   columns=list('AB'))
```

```
# doctest: +SKIP
A  B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```python
>>> df.cummax()  # doctest: +SKIP
A  B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```python
>>> df.cummax(axis=1)  # doctest: +SKIP
A  B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

`cummin(axis=None, skipna=True, out=None)`

Return cumulative minimum over a DataFrame or Series axis. This docstring was copied from pandas.core.frame.DataFrame.cummin.

Some inconsistencies with the Dask version may exist.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

**Parameters**
axis [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

*args, **kwargs: Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cummin [Series or DataFrame]

See also:

core.window.Expanding.min Similar functionality but ignores NaN values.

DataFrame.min Return the minimum over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```python
>>> s.cummin()  # doctest: +SKIP
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use skipna=False

```python
>>> s.cummin(skipna=False)  # doctest: +SKIP
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame
```python
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                   columns=list('AB'))
>>> df
A   B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'
```
```
```python
>>> df.cummin()
A   B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`
```
```
```python
>>> df.cummin(axis=1)
A   B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

cumprod (axis=None, skipna=True, dtype=None, out=None)

Return cumulative product over a DataFrame or Series axis.

This docstring was copied from pandas.core.frame.DataFrame.cumprod.

Some inconsistencies with the Dask version may exist.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

axis [{0 or ‘index’, 1 or ‘columns’}, default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

*args, **kwargs : Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns

cumprod [Series or DataFrame]

See also:

core.window.Expanding.prod Similar functionality but ignores NaN values.

DataFrame.prod Return the product over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.
**DataFrame.cumprod** Return cumulative product over DataFrame axis.

**Examples**

**Series**

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
>>> s
# doctest: +SKIP
0  2.0
1  NaN
2  5.0
3 -1.0
4  0.0
dtype: float64
```

By default, NA values are ignored.

```python
>>> s.cumprod()
# doctest: +SKIP
0  2.0
1  NaN
2  10.0
3 -10.0
4  -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```python
>>> s.cumprod(skipna=False)
# doctest: +SKIP
0  2.0
1  NaN
2  NaN
3  NaN
4  NaN
dtype: float64
```

**DataFrame**

```python
>>> df = pd.DataFrame([[2.0, 1.0], ...
... [3.0, np.nan], ...
... [1.0, 0.0]], ...
... columns=list('AB'))
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```python
>>> df.cumprod()
# doctest: +SKIP
   A  B
0  2.0  1.0
1  3.0 NaN
2  1.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`
```python
>>> df.cumprod(axis=1)  # doctest: +SKIP
   A  B
0  2.0  2.0
1  3.0    NaN
2  1.0    0.0
```

`cumsum(axis=None, skipna=True, dtype=None, out=None)`
Return cumulative sum over a DataFrame or Series axis.

This docstring was copied from pandas.core.frame.DataFrame.cumsum.

Some inconsistencies with the Dask version may exist.

Returns a DataFrame or Series of the same size containing the cumulative sum.

**Parameters**

- **axis** ([0 or ‘index’, 1 or ‘columns’], default 0] The index or the name of the axis. 0 is equivalent to None or ‘index’.
- **skipna** (boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- ***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

- **cumsum** [Series or DataFrame]

**See also:**

core.window.Expanding.sum Similar functionality but ignores NaN values.

DataFrame.sum Return the sum over DataFrame axis.

DataFrame.cummax Return cumulative maximum over DataFrame axis.

DataFrame.cummin Return cumulative minimum over DataFrame axis.

DataFrame.cumsum Return cumulative sum over DataFrame axis.

DataFrame.cumprod Return cumulative product over DataFrame axis.

**Examples**

**Series**

```python
>>> s = pd.Series([2, np.nan, 5, -1, 0])  # doctest: +SKIP
   # doctest: +SKIP
0   2.0
1  NaN
2   5.0
3  -1.0
4   0.0
dtype: float64
```

By default, NA values are ignored.
To include NA values in the operation, use `skipna=False`.

```python
>>> s.cumsum(skipna=False)  # doctest: +SKIP
0  2.0
1  NaN
2  NaN
3  NaN
4  NaN
dtype: float64
```

**DataFrame**

```python
>>> df = pd.DataFrame([[2.0, 1.0],
                     [3.0, np.nan],
                     [1.0, 0.0]],
                    columns=list('AB'))
>>> df  # doctest: +SKIP
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```python
>>> df.cumsum()  # doctest: +SKIP
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`.

```python
>>> df.cumsum(axis=1)  # doctest: +SKIP
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

desc__ibe__(split_every=False, percentiles=None, percentiles_method='default')

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

This docstring was copied from pandas.core.frame.DataFrame.describe.

Some inconsistencies with the Dask version may exist.

Currently, only numeric describe is supported.

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 (Not supported in Dask)] 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, (Not supported in Dask)] 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

Series or DataFrame Summary statistics of the Series or Dataframe provided.

See also:

- DataFrame.count Count number of non-NA/null observations.
- DataFrame.max Maximum of the values in the object.
- DataFrame.min Minimum of the values in the object.
- DataFrame.mean Mean of the values.
- DataFrame.std Standard deviation of the observations.
- DataFrame.select_dtypes Subset of a Dataframe including/excluding columns based on their dtype.

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.

```python
>>> s = pd.Series([1, 2, 3])  # doctest: +SKIP
>>> s.describe()  # doctest: +SKIP
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
dtype: float64
```

Describing a categorical Series.

```python
>>> s = pd.Series(['a', 'a', 'b', 'c'])  # doctest: +SKIP
>>> s.describe()  # doctest: +SKIP
count 4
unique 3
top a
freq 2
dtype: object
```

Describing a timestamp Series.

```python
>>> s = pd.Series([  # doctest: +SKIP
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])  # doctest: +SKIP
>>> s.describe()  # doctest: +SKIP
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.

```python
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),  # doctest: +SKIP
...                     'numeric': [1, 2, 3],
...                     'object': ['a', 'b', 'c']
... })
```

(continues on next page)
Describing all columns of a DataFrame regardless of data type.

```python
>>> df.describe(include='all') # doctest: +SKIP
categorical    numeric    object
  count          3          3          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.

```python
>>> df.numeric.describe() # doctest: +SKIP
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.

```python
>>> df.describe(include=[np.number]) # doctest: +SKIP
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.

```python
>>> df.describe(include=[np.object]) # doctest: +SKIP
object
```
Including only categorical columns from a DataFrame description.

```python
>>> df.describe(include=['category'])  # doctest: +SKIP
categorical
    count  unique top  freq
      3       3    c    1
```

Excluding numeric columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.number])  # doctest: +SKIP
categorical object
    count  unique top  freq
      3       3    f    1
```

Excluding object columns from a DataFrame description.

```python
>>> df.describe(exclude=[np.object])  # doctest: +SKIP
categorical numeric
    count  unique top   freq  mean  std  min  25%  50%  75%  max
      3       3    f  1.0    NaN  NaN  NaN  NaN  NaN  NaN  NaN
```

**diff** *(periods=1, axis=0)*

First discrete difference of element.

This docstring was copied from pandas.core.frame.DataFrame.diff.

Some inconsistencies with the Dask version may exist.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

**Parameters**

- **periods** : [int, default 1] Periods to shift for calculating difference, accepts negative values.
- **axis** : [0 or ‘index’, 1 or ‘columns’], default 0] Take difference over rows (0) or columns (1).

**Returns**
**diffed** [DataFrame]

See also:

*Series.diff* First discrete difference for a Series.

*DataFrame.pct_change* Percent change over given number of periods.

*DataFrame.shift* Shift index by desired number of periods with an optional time freq.

**Examples**

Difference with previous row

```python
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6], # doctest: +SKIP
... 'b': [1, 1, 2, 3, 5, 8],
... 'c': [1, 4, 9, 16, 25, 36]})

>>> df
give
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>8</td>
<td>36</td>
</tr>
</tbody>
</table>

>>> df.diff()
give
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>1.0</td>
<td>5.0</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
<td>1.0</td>
<td>7.0</td>
</tr>
<tr>
<td>4</td>
<td>1.0</td>
<td>2.0</td>
<td>9.0</td>
</tr>
<tr>
<td>5</td>
<td>1.0</td>
<td>3.0</td>
<td>11.0</td>
</tr>
</tbody>
</table>
```

Difference with previous column

```python
>>> df.diff(axis=1) # doctest: +SKIP
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>-1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>-1.0</td>
<td>7.0</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>-1.0</td>
<td>13.0</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
<td>0.0</td>
<td>20.0</td>
</tr>
<tr>
<td>5</td>
<td>NaN</td>
<td>2.0</td>
<td>28.0</td>
</tr>
</tbody>
</table>
```

Difference with 3rd previous row

```python
>>> df.diff(periods=3) # doctest: +SKIP
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>2.0</td>
<td>15.0</td>
</tr>
<tr>
<td>4</td>
<td>3.0</td>
<td>4.0</td>
<td>21.0</td>
</tr>
<tr>
<td>5</td>
<td>3.0</td>
<td>6.0</td>
<td>27.0</td>
</tr>
</tbody>
</table>
```

Difference with following row
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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:

Series.rtruediv

Examples

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd']) # doctest: +SKIP
>>> a  # doctest: +SKIP
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e']) # doctest: +SKIP
>>> b  # doctest: +SKIP
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
>>> a.add(b, fill_value=0) # doctest: +SKIP
a 2.0
b 1.0
c 1.0
d 1.0
(continues on next page)
drop_duplicates (split_every=None, split_out=1, **kwargs)

Return DataFrame with duplicate rows removed, optionally only considering certain columns.

This docstring was copied from pandas.core.frame.DataFrame.drop_duplicates.

Some inconsistencies with the Dask version may exist.

Parameters

subset [column label or sequence of labels, optional (Not supported in Dask)] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [‘first’, ‘last’, False], default ‘first’ (Not supported in Dask)]

• first : Drop duplicates except for the first occurrence.
• last : Drop duplicates except for the last occurrence.
• False : Drop all duplicates.

inplace [boolean, default False (Not supported in Dask)] Whether to drop duplicates in place or to return a copy.

Returns

deduplicated [DataFrame]

dropna ()

Return a new Series with missing values removed.

This docstring was copied from pandas.core.series.Series.dropna.

Some inconsistencies with the Dask version may exist.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

axis [{0 or ‘index’}, default 0 (Not supported in Dask)] There is only one axis to drop values from.

inplace [bool, default False (Not supported in Dask)] If True, do operation inplace and return None.

**kwargs Not in use.

Returns

Series Series with NA entries dropped from it.

See also:

Series.isna Indicate missing values.
Series.notna Indicate existing (non-missing) values.
Series.fillna Replace missing values.
DataFrame.dropna Drop rows or columns which contain NA values.
Index.dropna Drop missing indices.
Examples

```python
>>> ser = pd.Series([1., 2., np.nan])  # doctest: +SKIP
```
```
0   1.0  
1   2.0  
2   NaN  
```
dtype: float64

Drop NA values from a Series.

```python
>>> ser.dropna()  # doctest: +SKIP
```
```
0   1.0  
1   2.0  
```
dtype: float64

Keep the Series with valid entries in the same variable.

```python
>>> ser.dropna(inplace=True)  # doctest: +SKIP
```
```
0   1.0  
1   2.0  
```
dtype: float64

Empty strings are not considered NA values. None is considered an NA value.

```python
>>> ser = pd.Series([np.NaN, 2, pd.NaT, '', None, 'I stay'])  # doctest: +SKIP
```
```
0  NaN   
1   2  
2  NaT  
3   
4   None 
5   I stay 
```
dtype: object

```python
>>> ser.dropna()  # doctest: +SKIP
```
```
1   2  
3   
5   I stay 
```
dtype: object

```
dt

Namespace of datetime methods

dtype

Return data type

eq (other, level=none, fill_value=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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before
computation. If data in both corresponding Series locations is missing the result will be missing.

**level** [int or name] Broadcast across a level, matching Index values on the passed Multi
Index level

**Returns**

**result** [Series]

**See also:**

Series.None

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a  # doctest: +SKIP
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b  # doctest: +SKIP
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a  2.0
b  1.0
c  1.0
d  1.0
e  NaN
dtype: float64
```

**ffill**(axis=None, limit=None)

Synonym for **DataFrame.fillna()** with method='ffill'.

**fillna**(value=None, method=None, limit=None, axis=None)

Fill NA/NaN values using the specified method.

This docstring was copied from pandas.core.frame.DataFrame.fillna.

Some inconsistencies with the Dask version may exist.

**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.

axis  [[0 or ‘index’, 1 or ‘columns’]]

inplace [boolean, default False (Not supported in Dask)] 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 (Not supported in Dask)] 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:

interpolate Fill NaN values using interpolation.

reindex, asfreq

Examples

```python
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],  # doctest: +SKIP
...                     [3, 4, np.nan, 1],
...                     [np.nan, np.nan, np.nan, 5],
...                     [np.nan, 3, np.nan, 4]],
...                     columns=list('ABCD'))
>>> df  # doctest: +SKIP
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.

```python
>>> df.fillna(0)  # doctest: +SKIP
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.

```python
>>> df.fillna(method='ffill')  # doctest: +SKIP
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.

```python
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}  # doctest: +SKIP
>>> df.fillna(value=values)  # doctest: +SKIP
      A  B  C  D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  2.0  1.0
2  0.0  1.0  2.0  5.0
3  0.0  3.0  2.0  4.0
```

Only replace the first NaN element.

```python
>>> df.fillna(value=values, limit=1)  # doctest: +SKIP
      A  B  C  D
0  0.0  2.0  2.0  0.0
1  3.0  4.0  NaN  1.0
2  NaN  1.0  NaN  5.0
3  NaN  3.0  NaN  4.0
```

`first` *(offset)*

Convenience method for subsetting initial periods of time series data based on a date offset.

This docstring was copied from pandas.core.frame.DataFrame.first.

Some inconsistencies with the Dask version may exist.

**Parameters**

- offset  [string, DateOffset, dateutil.relativedelta]

**Returns**

- subset  [same type as caller]

**Raises**

- TypeError  If the index is not a DatetimeIndex

**See also:**

- `last`  Select final periods of time series based on a date offset.
- `at_time`  Select values at a particular time of the day.
- `between_time`  Select values between particular times of the day.

**Examples**

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')  # doctest: +SKIP
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)  # doctest: +SKIP
>>> ts  # doctest: +SKIP
     A
2018-04-09 1
2018-04-11 2
2018-04-13 3
2018-04-15 4
```

Get the rows for the first 3 days:
Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing.
level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:
Series.rfloordiv

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a
a    1.0
b    1.0
c    1.0
d   NaN
dtype: float64
```
```
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
```
```
>>> b
a    1.0
b   NaN
d    1.0
e   NaN
dtype: float64
```
```
>>> a.add(b, fill_value=0)  # doctest: +SKIP
```
```
a    2.0
b    1.0
c    1.0
d    1.0
e   NaN
dtype: float64
```
**ge** *(other, level=None, fill_value=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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

- **result** [Series]

**See also:**

Series.None

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest:+SKIP
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest:+SKIP
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

**get_partition** *(n)*

Get a dask DataFrame/Series representing the `nth` partition.

**groupby** *(by=None, **kwargs)*

Group DataFrame or Series using a mapper or by a Series of columns.

This docstring was copied from pandas.core.series.Series.groupby.

Some inconsistencies with the Dask version may exist.
A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

**Parameters**

- **by** [mapping, function, label, or list of labels] 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 label or list of labels may be passed to group by the columns in self. Notice that a tuple is interpreted a (single) key.

- **axis** [{0 or ‘index’, 1 or ‘columns’}, default 0 (Not supported in Dask)] Split along rows (0) or columns (1).

- **level** [int, level name, or sequence of such, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

- **as_index** [bool, default True (Not supported in Dask)] 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** [bool, default True (Not supported in Dask)] 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** [bool, default True (Not supported in Dask)] When calling apply, add group keys to index to identify pieces.

- **squeeze** [bool, default False (Not supported in Dask)] Reduce the dimensionality of the return type if possible, otherwise return a consistent type.

- **observed** [bool, default False (Not supported in Dask)] This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

  New in version 0.23.0.

- ****kwarg**s Optional, only accepts keyword argument ‘mutated’ and is passed to groupby.

**Returns**

- **DataFrameGroupBy or SeriesGroupBy** Depends on the calling object and returns groupby object that contains information about the groups.

**See also:**

- **resample** Convenience method for frequency conversion and resampling of time series.

**Notes**

See the user guide for more.

**Examples**
Hierarchical Indexes

We can groupby different levels of a hierarchical index using the `level` parameter:

```python
gt (other, level=None, fill_value=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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is 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

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a
# doctest: +SKIP
a    1.0
b    1.0
c    1.0
d   NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b
# doctest: +SKIP
a   1.0
b   NaN
d   1.0
e   NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a    2.0
b    1.0
c    1.0
d    1.0
e   NaN
dtype: float64
```

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.

This docstring was copied from pandas.core.frame.DataFrame.idxmax. Some inconsistencies with the Dask version may exist.

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. This docstring was copied from pandas.core.frame.DataFrame.idxmin.
Some inconsistencies with the Dask version may exist.

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

isin (values)
Check whether values are contained in Series.
This docstring was copied from pandas.core.series.Series.isin.
Some inconsistencies with the Dask version may exist.
Return a boolean Series showing whether each element in the Series matches an element in the passed sequence of `values` exactly.

**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:**

- `DataFrame.isin` Equivalent method on DataFrame.

**Examples**

```python
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
# doctest: +SKIP
   0     True
   1     True
   2     True
   3    False
   4     True
   5    False
Name: animal, dtype: bool
```

Passing a single string as `s.isin('lama')` will raise an error. Use a list of one element instead:

```python
>>> s.isin(['lama'])
# doctest: +SKIP
   0     True
   1    False
   2     True
   3    False
   4     True
   5    False
Name: animal, dtype: bool
```

`isna()`

Detect missing values.

This docstring was copied from pandas.core.frame.DataFrame.isna.

Some inconsistencies with the Dask version may exist.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None` or `numpy.NaN`, gets mapped to `True` values. Everything else gets mapped to `False` values. Characters such as empty strings ' ' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns**
DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.isnull Alias of isna.
DataFrame.notna Boolean inverse of isna.
DataFrame.dropna Omit axes labels with missing values.
isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```python
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
                              pd.Timestamp('1940-04-25')],
                     'name': ['Alfred', 'Batman', ''],
                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
    age      born       name    toy
0  5.00 NaT       Alfred    None
1  6.00 1939-05-27  Batman  Batmobile
2 NaN  1940-04-25  Joker   
```

```python
>>> df.isna()
    age      born       name    toy
0  False  True  False  True
1  False  False  False  False
2  True  False  False  False
```

Show which entries in a Series are NA.

```python
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0 5.0
1 6.0
2 NaN
dtype: float64
```

```python
>>> ser.isna()
0  False
1  False
2  True
dtype: bool
```

isnull ()

Detect missing values.

This docstring was copied from pandas.core.frame.DataFrame.isnull.

Some inconsistencies with the Dask version may exist.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty.
strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

DataFrame.isnull Alias of isna.
DataFrame.notna Boolean inverse of isna.
DataFrame.dropna Omit axes labels with missing values.
isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```python
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                   'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                            pd.Timestamp('1940-04-25')],
...                   'name': ['Alfred', 'Batman', ''],
...                   'toy': [None, 'Batmobile', 'Joker']})
```

```python
>>> df
   age   born            name   toy
0  5.0    NaT   Alfred    None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25    Joker          
```

```python
>>> df.isna()
   age   born            name   toy
0 False  True    False    True
1 False  False    False    False
2  True  False    False    False
```

Show which entries in a Series are NA.

```python
>>> ser = pd.Series([5, 6, np.NaN])
```

```python
>>> ser
0   5.0
1   6.0
2  NaN
dtype: float64
```

```python
>>> ser.isna()
0    False
1    False
2    True
dtype: bool
```

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.
This docstring was copied from pandas.core.frame.DataFrame.last.
Some inconsistencies with the Dask version may exist.

Parameters
offset [string, DateOffset, dateutil.relativedelta]

Returns
subset [same type as caller]

Raises
TypeError If the index is not a DatetimeIndex

See also:
first Select initial periods of time series based on a date offset.
at_time Select values at a particular time of the day.
between_time Select values between particular times of the day.

Examples

```python
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D') # doctest: +SKIP
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i) # doctest: +SKIP
>>> ts
    A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```
Get the rows for the last 3 days:

```python
>>> ts.last('3D') # doctest: +SKIP
     A
2018-04-13  3
2018-04-15  4
```
Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

le (other, level=None, fill_value=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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
**level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

**result** [Series]

**See also:**

Series.None

**Examples**

```python
def add(b, fill_value=0) # doctest: +SKIP
da 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```

**loc**

Purely label-location based indexer for selection by label.

```python
def.loc["b"] # doctest: +SKIP
def.loc["b":"d"] # doctest: +SKIP
```

**lt** (other, 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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

4.9. DataFrame
Returns

result [Series]

See also:

Series.None

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest:
˓
...
>>> a # doctest: +SKIP
a    1.0
b    1.0
c    1.0
d   NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  #
˓
...
>>> b  # doctest: +SKIP
a    1.0
b   NaN
d    1.0
e   NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a    2.0
b    1.0
c    1.0
d    1.0
e   NaN
dtype: float64
```

`map(arg, na_action=None, meta='__no_default__')`

Map values of Series according to input correspondence.

This docstring was copied from pandas.core.series.Series.map.

Some inconsistencies with the Dask version may exist.

Used for substituting each value in a Series with another value, that may be derived from a function, a
dict or a Series.

Parameters

arg [function, dict, or Series] Mapping correspondence.

na_action [{None, ‘ignore’}, default None] If ‘ignore’, propagate NaN values, without
passing them to the mapping correspondence.

DataFrame or pd.Series that matches the dtypes and column names of the out-
put. 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

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.

Examples

```python
>>> s = pd.Series(['cat', 'dog', np.nan, 'rabbit'])  # doctest: +SKIP
>>> s  # doctest: +SKIP
0   cat
1   dog
2  NaN
3  rabbit
dtype: object
```

map accepts a dict or a Series. Values that are not found in the dict are converted to NaN, unless the dict has a default value (e.g. defaultdict):

```python
>>> s.map({'cat': 'kitten', 'dog': 'puppy'})  # doctest: +SKIP
0   kitten
1   puppy
2   NaN
3   NaN
dtype: object
```

It also accepts a function:

```python
>>> s.map('I am a {}'.format)  # doctest: +SKIP
0  I am a cat
1  I am a dog
2  I am a nan
3  I am a rabbit
dtype: object
```

To avoid applying the function to missing values (and keep them as NaN) na_action='ignore' can be used:

```python
>>> s.map('I am a {}'.format, na_action='ignore')  # doctest: +SKIP
0  I am a cat
1  I am a dog
2   NaN
3  I am a rabbit
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:

```python
>>> 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():

```python
```
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:

```python
def diff(df, periods=1):
    ... before, after = (periods, 0) if periods > 0 else (0, -periods)
    ... return df.map_overlap(lambda df, periods=1: df.diff(periods),
                              periods, 0, periods=periods)
```

```bash
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 `pd.Timedelta` for time-based windows.

```bash
ts = pd.Series(range(10), index=pd.date_range('2017', periods=10))
```

```bash
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. Arguments and keywords may contain
Scalar, Delayed or regular python objects. DataFrame-like args (both dask and
pandas) will be repartitioned to align (if necessary) before applying the function.

DataFrame or pd.Series that matches the dtypes and column names of the out-
put. 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.

Examples

Given a DataFrame, Series, or Index, such as:

```python
>>> import dask.dataframe as dd
>>> df = pd.DataFrame({'x': [1, 2, 3, 4, 5],
...                    'y': [1., 2., 3., 4., 5.1])
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> ddf.map_partitions(func).clear_divisions()  # doctest: +SKIP
```

**mask (cond, other=nan)**

Replace values where the condition is True.

This docstring was copied from pandas.core.frame.DataFrame.mask.

Some inconsistencies with the Dask version may exist.

**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 (Not supported in Dask)] Whether to perform the operation in place on the data.

- **axis** [int, default None (Not supported in Dask)] Alignment axis if needed.

- **level** [int, default None (Not supported in Dask)] Alignment level if needed.

- **errors** [str, {‘raise’, ‘ignore’}, default raise (Not supported in Dask)] Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

  - `raise`: allow exceptions to be raised.
  - `ignore`: suppress exceptions. On error return original object.

- **try_cast** [boolean, default False (Not supported in Dask)] Try to cast the result back to the input type (if possible).

- **raise_on_error** [boolean, default True (Not supported in Dask)] Whether to raise on invalid data types (e.g. trying to where on strings).

  Deprecated since version 0.21.0: Use `errors`.

**Returns**

- **wh** [same type as caller]

See also:

- `DataFrame.where()` Return an object of same shape as self.
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

```python
>>> s = pd.Series(range(5))  # doctest: +SKIP
>>> s.where(s > 0)  # doctest: +SKIP
    0   NaN
    1   1
    2   2
    3   3
    4   4
  dtype: float64

>>> s.mask(s > 0)  # doctest: +SKIP
    0   NaN
    1   NaN
    2   NaN
    3   NaN
    4   NaN
  dtype: float64

>>> s.where(s > 1, 10)  # doctest: +SKIP
    0   10
    1   10
    2    2
    3    3
    4    4
  dtype: int64

>>> m = df % 3 == 0  # doctest: +SKIP
>>> df.where(m, -df)  # doctest: +SKIP
    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)  # doctest: +SKIP
    A    B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

(continues on next page)
max (axis=None, skipna=True, split_every=False, out=None)

Return the maximum of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.max.

Some inconsistencies with the Dask version may exist.

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)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None (Not supported in Dask)] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

max [Series or DataFrame (if level specified)]

See also:

Series.sum Return the sum.
Series.min Return the minimum.
Series.max Return the maximum.
Series.idxmin Return the index of the minimum.
Series.idxmax Return the index of the maximum.
DataFrame.min Return the sum over the requested axis.
DataFrame.min Return the minimum over the requested axis.
DataFrame.max Return the maximum over the requested axis.
DataFrame.idxmin Return the index of the minimum over the requested axis.
DataFrame.idxmax Return the index of the maximum over the requested axis.
Examples

```python
>>> idx = pd.MultiIndex.from_arrays([['warm', 'warm', 'cold', 'cold'], ['dog', 'falcon', 'fish', 'spider'], names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)  # doctest: +SKIP
>>> s
blooded  animal
    warm  dog  4
       falcon  2
    cold  fish  0
       spider  8
Name: legs, dtype: int64

>>> s.max()  # doctest: +SKIP
8

Max using level names, as well as indices.

```python
>>> s.max(level='blooded')  # doctest: +SKIP
blooded
    warm  4
    cold  8
Name: legs, dtype: int64

>>> s.max(level=0)  # doctest: +SKIP
blooded
    warm  4
    cold  8
Name: legs, dtype: int64
```

`mean` *(axis=None, skipna=True, split_every=False, dtype=None, out=None)*

Return the mean of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.mean.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **axis** [[index (0), columns (1)]] Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** [bool, default None (Not supported in Dask)] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs Additional keyword arguments to be passed to the function.

**Returns**

- **mean** [Series or DataFrame (if level specified)]

**memory_usage** *(index=True, deep=False)*

Return the memory usage of the Series.
This docstring was copied from pandas.core.series.Series.memory_usage.
Some inconsistencies with the Dask version may exist.

The memory usage can optionally include the contribution of the index and of elements of object dtype.

Parameters

   index [bool, default True] Specifies whether to include the memory usage of the Series
          index.

   deep [bool, default False] If True, introspect the data deeply by interrogating object
          dtypes for system-level memory consumption, and include it in the returned value.

Returns

   int  Bytes of memory consumed.

See also:

numpy.ndarray.nbytes  Total bytes consumed by the elements of the array.

DataFrame.memory_usage  Bytes consumed by a DataFrame.

Examples

```python
>>> s = pd.Series(range(3))  # doctest: +SKIP
>>> s.memory_usage()  # doctest: +SKIP
104

Not including the index gives the size of the rest of the data, which is necessarily smaller:

```python
>>> s.memory_usage(index=False)  # doctest: +SKIP
24

The memory footprint of object values is ignored by default:

```python
>>> s = pd.Series(["a", "b"])  # doctest: +SKIP
>>> s.values  # doctest: +SKIP
array(['a', 'b'], dtype=object)
>>> s.memory_usage()  # doctest: +SKIP
96
>>> s.memory_usage(deep=True)  # doctest: +SKIP
212
```

min(axis=None, skipna=True, split_every=False, out=None)

Return the minimum of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.min.
Some inconsistencies with the Dask version may exist.

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)] Axis for the function to be applied on.

   skipna [bool, default True] Exclude NA/null values when computing the result.
level [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None (Not supported in Dask)] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**kwargs Additional keyword arguments to be passed to the function.

Returns

min [Series or DataFrame (if level specified)]

See also:

Series.sum Return the sum.
Series.min Return the minimum.
Series.max Return the maximum.
Series.idxmin Return the index of the minimum.
Series.idxmax Return the index of the maximum.
DataFrame.min Return the sum over the requested axis.
DataFrame.min Return the minimum over the requested axis.
DataFrame.max Return the maximum over the requested axis.
DataFrame.idxmin Return the index of the minimum over the requested axis.
DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```python
>>> idx = pd.MultiIndex.from_arrays([... # doctest: +SKIP
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx) # doctest: +SKIP
>>> s
blooded animal
dog    4
falcon  2
fish    0
spider  8
Name: legs, dtype: int64

>>> s.min() # doctest: +SKIP
0

Min using level names, as well as indices.

```
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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is 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

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```
mul \((other, level=None, fill_value=None, axis=0)\)
Multiplication of series and other, element-wise (binary operator \(mul\)).

Equivalent to \(series \times other\), 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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

- **result** [Series]

**See also:**

- `Series.rmul`

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest:
˓→+SKIP
>>> a  # doctest: +SKIP
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
˓→+SKIP
>>> b  # doctest: +SKIP
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a  2.0
b  1.0
c  1.0
d  1.0
e  NaN
dtype: float64
```

**nbytes**

- Number of bytes

**ndim**

- Return dimensionality

**ne \((other, level=None, fill_value=None, axis=0)\)**

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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is 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

**Examples**

```python
given some data:

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a
a    1.0
b    1.0
c    1.0
d   NaN
dtype: float64

>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b
a    1.0
b   NaN
d    1.0
e   NaN
dtype: float64

Result:

>>> a.add(b, fill_value=0)  # doctest: +SKIP
a    2.0
b    1.0
c    1.0
d    1.0
e   NaN
dtype: float64
```

**nlargest** *(n=5, split_every=None)*

Return the largest `n` elements.

This docstring was copied from pandas.core.series.Series.nlargest.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **n** [int, default 5] Return this many descending sorted values.
keep  [{‘first’, ‘last’, ‘all’}, default ‘first’ (Not supported in Dask)] When there are duplicate values that cannot all fit in a Series of \( n \) elements:

- **first**: take the first occurrences based on the index order
- **last**: take the last occurrences based on the index order
- **all** [keep all occurrences. This can result in a Series of] size larger than \( n \).

**Returns**

**Series** The \( n \) largest values in the Series, sorted in decreasing order.

**See also:**

- **Series.nsmallest** Get the \( n \) smallest elements.
- **Series.sort_values** Sort Series by values.
- **Series.head** Return the first \( n \) rows.

**Notes**

Faster than `.sort_values(ascending=False).head(n)` for small \( n \) relative to the size of the Series object.

**Examples**

```python
countries_population = {"Italy": 59000000, "France": 65000000, #
doctest: +SKIP
...
   "Malta": 434000, "Maldives": 434000,
...
   "Brunei": 434000, "Iceland": 337000,
...
   "Nauru": 11300, "Tuvalu": 11300,
...
   "Anguilla": 11300, "Monserat": 5200}
>>> s = pd.Series(countries_population)  # doctest: +SKIP
>>> s
Italy       59000000
France      65000000
Malta       434000
Maldives    434000
Brunei      434000
Iceland     337000
Nauru       11300
Tuvalu      11300
Anguilla    11300
Monserat    5200
dtype: int64
```

The \( n \) largest elements where \( n=5 \) by default.

```python
>>> s.nlargest()   # doctest: +SKIP
France       65000000
Italy        59000000
Malta        434000
Maldives     434000
Brunei       434000
dtype: int64
```

The \( n \) largest elements where \( n=3 \). Default `keep` value is ‘first’ so Malta will be kept.
```python
>>> s.nlargest(3)  # doctest: +SKIP
 France  65000000
 Italy   59000000
 Malta   434000
 dtype: int64
```

The \( n \) largest elements where \( n=3 \) and keeping the last duplicates. Brunei will be kept since it is the last with value 434000 based on the index order.

```python
>>> s.nlargest(3, keep='last')  # doctest: +SKIP
 France  65000000
 Italy   59000000
 Brunei  434000
 dtype: int64
```

The \( n \) largest elements where \( n=3 \) with all duplicates kept. Note that the returned Series has five elements due to the three duplicates.

```python
>>> s.nlargest(3, keep='all')  # doctest: +SKIP
 France  65000000
 Italy   59000000
 Malta   434000
 Maldives 434000
 Brunei  434000
 dtype: int64
```

`notnull()`

Detect existing (non-missing) values.

This docstring was copied from pandas.core.frame.DataFrame.notnull.

Some inconsistencies with the Dask version may exist.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

**Returns**

- **DataFrame** Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

**See also:**

- **DataFrame.notnull** Alias of notna.
- **DataFrame.isna** Boolean inverse of notna.
- **DataFrame.dropna** Omit axes labels with missing values.
- **notna** Top-level notna.

**Examples**

Show which entries in a DataFrame are not NA.
Show which entries in a Series are not NA.

```python
>>> ser = pd.Series([5, 6, np.NaN])  # doctest: +SKIP
... # doctest: +SKIP
>>> ser
0  5.0
1  6.0
2 NaN
dtype: float64

>>> ser.notna()  # doctest: +SKIP
0  True
1  True
2 False
dtype: bool
```

### npartitions
Return number of partitions

### nsmallest
```
(n=5, split_every=None)
Return the smallest n elements.
```

This docstring was copied from pandas.core.series.Series.nsmallest. Some inconsistencies with the Dask version may exist.

**Parameters**

- **n** [int, default 5] Return this many ascending sorted values.
- **keep** [{‘first’, ‘last’, ‘all’}, default ‘first’ (Not supported in Dask)] When there are duplicate values that cannot all fit in a Series of n elements:
  - first: take the first occurrences based on the index order
  - last: take the last occurrences based on the index order
  - all [keep all occurrences. This can result in a Series of] size larger than n.

**Returns**

- **Series** The n smallest values in the Series, sorted in increasing order.

**See also:**
**Series.nlargest** Get the $n$ largest elements.

**Series.sort_values** Sort Series by values.

**Series.head** Return the first $n$ rows.

**Notes**

Faster than `.sort_values().head(n)` for small $n$ relative to the size of the Series object.

**Examples**

```python
>>> countries_population = {
    "Italy": 59000000, "France": 65000000, 
    # doctest: +SKIP
    ...,
    "Brunei": 434000, "Malta": 434000,
    ...,
    "Maldives": 434000, "Iceland": 337000,
    ...,
    "Nauru": 11300, "Tuvalu": 11300,
    ...,
    "Anguilla": 11300, "Monserat": 5200
}

>>> s = pd.Series(countries_population)  # doctest: +SKIP

>>> s  # doctest: +SKIP
Italy    59000000
France   65000000
Brunei   434000
Malta    434000
Maldives 434000
Iceland  337000
Nauru    11300
Tuvalu   11300
Anguilla 11300
Monserat 5200
dtype: int64

The $n$ largest elements where $n=5$ by default.

```python

```python
>>> s.nlargest()  # doctest: +SKIP
Monserat    5200
Nauru        11300
Tuvalu       11300
Anguilla     11300
Iceland      337000
dtype: int64
```

The $n$ smallest elements where $n=3$. Default `keep` value is ‘first’ so Nauru and Tuvalu will be kept.

```python
>>> s.nsmallest(3)  # doctest: +SKIP
Monserat    5200
Nauru       11300
Tuvalu      11300
dtype: int64
```

The $n$ smallest elements where $n=3$ and keeping the last duplicates. Anguilla and Tuvalu will be kept since they are the last with value 11300 based on the index order.

```python
>>> s.nsmallest(3, keep='last')  # doctest: +SKIP
Monserat    5200
Anguilla    11300
```

(continues on next page)
The $n$ smallest elements where $n=3$ with all duplicates kept. Note that the returned Series has four elements due to the three duplicates.

```python
>>> s.nsmallest(3, keep='all')  # doctest: +SKIP
Monserat  5200
Nauru     11300
Tuvalu    11300
Anguilla  11300
```

def nunique(split_every=None)

Return number of unique elements in the object.

This docstring was copied from pandas.core.series.Series.nunique.

Some inconsistencies with the Dask version may exist.

Excludes NA values by default.

Parameters

- **dropna** [boolean, default True (Not supported in Dask)] Don’t include NaN in the count.

Returns

nunique [int]

def 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

def partitions

Slice dataframe by partitions

This allows partitionwise slicing of a Dask Dataframe. You can perform normal Numpy-style slicing but now rather than slice elements of the array you slice along partitions so, for example, `df.partitions[:5]` produces a new Dask Dataframe of the first five partitions.

Returns

A Dask DataFrame

Examples

```python
>>> df.partitions[0]  # doctest: +SKIP
>>> df.partitions[:3] # doctest: +SKIP
>>> df.partitions[::10]  # doctest: +SKIP
```
**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.

The action of function differs significantly depending on the active task scheduler. If the task scheduler supports asynchronous computing, such as is the case of the dask.distributed scheduler, then persist will return immediately and the return value’s task graph will contain Dask Future objects. However if the task scheduler only supports blocking computation then the call to persist will block and the return value’s task graph will contain concrete Python results.

This function is particularly useful when using distributed systems, because the results will be kept in distributed memory, rather than returned to the local process as with compute.

**Parameters**

- **scheduler** [string, optional] Which scheduler to use like “threads”, “synchronous” or “processes”. 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 function.

**Returns**

New dask collections backed by in-memory data

**See also:**

dask.base.persist

**pipe**(func, *args, **kwargs)

Apply func(self, *args, **kwargs).

This docstring was copied from pandas.core.frame.DataFrame.pipe.

Some inconsistencies with the Dask version may exist.

**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:**

DataFrame.apply, DataFrame.applymap, 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)  # doctest: +SKIP

You can write

```python
>>> (df.pipe(h)  # doctest: +SKIP
...     .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:

```python
>>> (df.pipe(h)  # doctest: +SKIP
...     .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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing.
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

- **result** [Series]

**See also:**

Series `rpow`

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a  # doctest: +SKIP
a 1.0  
b 1.0  
c 1.0  
d NaN  
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b  # doctest: +SKIP
a 1.0
b NaN
```
dask Documentation, Release 1.2.2

(continued from previous page)

```
d 1.0
e NaN
dtype: float64
```

```python
>>> a.add(b, fill_value=0)  # doctest: +SKIP
```

```
a 2.0
b 1.0
c 1.0
d 1.0
e  NaN
dtype: float64
```

```python
prod (axis=None, skipna=True, split_every=False, dtype=None, out=None, min_count=None)
```

Return the product of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.prod.

Some inconsistencies with the Dask version may exist.

**Parameters**

axis [{index (0), columns (1)}] Axis for the function to be applied on.

skipna [bool, default True] Exclude NA/null values when computing the result.

level [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

numeric_only [bool, default None (Not supported in Dask)] 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 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

**kwargs Additional keyword arguments to be passed to the function.

**Returns**

prod [Series or DataFrame (if level specified)]

**Examples**

By default, the product of an empty or all-NA Series is 1

```python
>>> pd.Series([]).prod()  # doctest: +SKIP
1.0
```

This can be controlled with the min_count parameter

```python
>>> pd.Series([]).prod(min_count=1)  # doctest: +SKIP
nan
```

Thanks to the skipna parameter, min_count handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).prod()  # doctest: +SKIP
1.0
```
quantile \( (q=0.5, \text{method}='default') \)
Approximate quantiles of Series

Parameters

- q [list/array of floats, default 0.5 (50%)] Iterable of numbers ranging from 0 to 1 for the desired quantiles
- method [[{'default', 'tdigest', 'dask'}, optional] What method to use. By default will use dask’s internal custom algorithm (‘dask’). If set to ‘tdigest’ will use tdigest for floats and ints and fallback to the ‘dask’ otherwise.

radd \( (\text{other}, \text{level}=None, \text{fill_value}=None, \text{axis}=0) \)
Addition of series and other, element-wise (binary operator radd).
Equivalent to \text{other} + \text{series}, but with support to substitute a \text{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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
- level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns

result [Series]

See also:
Series.add

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a  # doctest: +SKIP
a    1.0
b    1.0
c    1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b  # doctest: +SKIP
a    1.0
b  NaN
d    1.0
e  NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
```

(continues on next page)
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

```python
>>> a, b = df.random_split([0.5, 0.5])  # doctest: +SKIP
```

80/10/10 split, consistent random_state

```python
>>> a, b, c = df.random_split([0.8, 0.1, 0.1], random_state=123)  # doctest: +SKIP
```

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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

- **result** [Series]

**See also:**

Series.truediv
Examples

```python
c>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
c>> a  # doctest: +SKIP
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
c>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
c>> b  # doctest: +SKIP
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
c>> a.add(b, fill_value=0)  # doctest: +SKIP
c   a  2.0
   b  1.0
   c  1.0
d  1.0
e   NaN
dtype: float64
```

reduction

```python
reduction(chunk, aggregate=None, combine=None, meta=None, 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.

kwars : All remaining keywords will be passed to chunk, combine, and aggregate.

Examples

```python
>>> 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:

```python
>>> res = ddf.reduction(lambda x: x.count(),
... aggregate=lambda x: x.sum())
```

```python
>>> 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).

```python
>>> def count_greater(x, value=0):
... return (x >= value).sum()
```

```python
>>> res = ddf.x.reduction(count_greater, aggregate=lambda x: x.sum(),
... chunk_kwargs={'value': 25})
```

```python
>>> res.compute()
25
```

Aggregate both the sum and count of a Series at the same time:

```python
>>> def sum_and_count(x):
... return pd.Series({'count': x.count(), 'sum': x.sum()},
... index=['count', 'sum'])
```

```python
>>> res = ddf.x.reduction(sum_and_count, aggregate=lambda x: x.sum())
```

```python
>>> 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.

```python
def sum_and_count(x):
    return pd.DataFrame({'count': x.count(), 'sum': x.sum()},
                        columns=['count', 'sum'])
```

```python
res = ddf.reduction(sum_and_count,
... aggregate=lambda x: x.groupby(level=0).sum())
```

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>50</td>
<td>1225</td>
</tr>
<tr>
<td>y</td>
<td>50</td>
<td>3725</td>
</tr>
</tbody>
</table>

`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 in-place.

- **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. 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

```python
>>> df = df.repartition(npartitions=10)  # doctest: +SKIP
>>> df = df.repartition(divisions=[0, 5, 10, 20])  # doctest: +SKIP
>>> df = df.repartition(freq='7d')  # doctest: +SKIP
```

`replace` *(to_replace=None, value=None, regex=False)*

Replace values given in `to_replace` with `value`.

This docstring was copied from pandas.core.frame.DataFrame.replace.

Some inconsistencies with the Dask version may exist.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

Parameters

`to_replace` [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to `to_replace` will be replaced with `value`
  - str: string exactly matching `to_replace` will be replaced with `value`
  - regex: regexs matching `to_replace` will be replaced with `value`

- list of str, regex, or numeric:
  - First, if `to_replace` and `value` are both lists, they must be the same length.
  - Second, if `regex=True` then all of the strings in both lists will be interpreted as regexes otherwise they will match directly. This doesn’t matter much for `value` since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.

- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, ("a": "b", "y": "z") replaces the value ‘a’ with ‘b’ and ‘y’ with ‘z’. To use a dict in this way the `value` parameter should be `None`.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, ("a": 1, "b": "z") looks for the value 1 in column ‘a’ and the value ‘z’ in column ‘b’ and replaces these values with whatever is specified in `value`. The `value` parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., {'a': {'b': np.nan}}, are read as follows: look in column ‘a’ for the value ‘b’ and replace it with NaN. The `value` parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) cannot be regular expressions.

- None:
- This means that the regex argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If value is also None then this must be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching to_replace with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [bool, default False (Not supported in Dask)] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None (Not supported in Dask)] Maximum size gap to forward or backward fill.

regex [bool or same types as to_replace, default False] Whether to interpret to_replace and/or value as regular expressions. If this is True then to_replace must be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case to_replace must be None.

method [{‘pad’, ‘ffill’, ‘bfill’, None} (Not supported in Dask)] The method to use when for replacement, when to_replace is a scalar, list or tuple and value is None.

Changed in version 0.23.0: Added to DataFrame.

Returns

DataFrame Object after replacement.

Raises

AssertionError

- If regex is not a bool and to_replace is not None.

TypeError

- If to_replace is a dict and value is not a list, dict, ndarray, or Series
- If to_replace is None and regex is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple bool or datetime64 objects and the arguments to to_replace does not match the type of the value being replaced

ValueError

- If a list or an ndarray is passed to to_replace and value but they are not the same length.

See also:

DataFrame.fillna Fill NA values.

DataFrame.where Replace values based on boolean condition.

Series.str.replace Simple string replacement.
Notes

- Regex substitution is performed under the hood with \texttt{re.sub}. The rules for substitution for \texttt{re.sub} are the same.

- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtypes to be matched. However, if those floating point numbers are strings, then you can do this.

- This method has a lot of options. You are encouraged to experiment and play with this method to gain intuition about how it works.

- When dict is used as the \texttt{to_replace} value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Examples

Scalar ‘to_replace’ and ‘value’

```python
>>> s = pd.Series([0, 1, 2, 3, 4])  # doctest: +SKIP
>>> s.replace(0, 5)  # doctest: +SKIP
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```python
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
                   'B': [5, 6, 7, 8, 9],
                   'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)  # doctest: +SKIP
     A  B  C
0   5  5  a
1   1  6  b
2   2  7  c
3   3  8  d
4   4  9  e
```

List-like ‘to_replace’

```python
>>> df.replace([0, 1, 2, 3], 4)  # doctest: +SKIP
     A  B  C
0   4  5  a
1   4  6  b
2   4  7  c
3   4  8  d
4   4  9  e
```

```python
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])  # doctest: +SKIP
     A  B  C
0   4  5  a
1   3  6  b
2   2  7  c
3   1  8  d
4   4  9  e
```
```python
>>> s.replace([1, 2], method='bfill')  # doctest: +SKIP
0    0
1    3
2    3
3    3
4    4
dtype: int64
dict-like 'to_replace'

>>> df.replace({0: 10, 1: 100})  # doctest: +SKIP
   A  B  C
0  10  5  a
1  100 6  b
2    2 7  c
3    3 8  d
4    4 9  e

>>> df.replace({'A': {0: 100, 4: 400}})  # doctest: +SKIP
   A  B  C
0  100 5  a
1   1 6  b
2    2 7  c
3    3 8  d
4  400 9  e
```

Regular expression 'to_replace'

```python
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],  # doctest: +SKIP...
                     'B': ['abc', 'bar', 'xyz']})
...  # doctest: +SKIP
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)  # doctest: +SKIP
   A  B
0 new abc
1 foo new
2 bait xyz
```

```python
>>> df.replace(r'^ba.$', value='new')  # doctest: +SKIP
   A  B
0 new abc
1 foo new
2 bait xyz
```

```python
>>> df.replace(regex=r'^ba.$', value='new')  # doctest: +SKIP
   A  B
0 new abc
1 foo new
2 bait xyz
```

```python
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)  # doctest: +SKIP
   A  B
0 new abc
1 foo bar
2 bait xyz
```

```python
>>> df.replace(regex=r'^ba.$', value='new')  # doctest: +SKIP
   A  B
0 new abc
1 foo new
2 bait xyz
```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})  # doctest: +SKIP
   A    B
0  new  abc
1  xyz  new
2     bait  xyz

>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')  # doctest: +SKIP
   A    B
0  new  abc
1  new  new
2     bait  xyz

Note that when replacing multiple bool or datetime64 objects, the data types in the to_replace parameter must match the data type of the value being replaced:

```python
>>> df = pd.DataFrame({'A': [True, False, True],  # doctest: +SKIP
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False})  # raises # doctest: +SKIP
Traceback (most recent call last):
  ...TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a TypeError because one of the dict keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the to_replace parameter:

```python
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])  # doctest: +SKIP
>>> s.replace({'a': None})  # doctest: +SKIP
0    10
1   None
2   None
3      b
4   None
dtype: object
```

When one uses a dict as the to_replace value, it is like the value(s) in the dict are equal to the value parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None):

```python
>>> s.replace({'a': None})  # doctest: +SKIP
0    10
1   None
2   None
3      b
4   None
dtype: object
```

When value=None and to_replace is a scalar, list or tuple, replace uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```python
>>> s.replace('a', None)  # doctest: +SKIP
0    10
1    10
2      b
3      b
dtype: object
```

resample (rule, closed=None, label=None)
Resample time-series data.

4.9. DataFrame 651
This docstring was copied from pandas.core.frame.DataFrame.resample.

Some inconsistencies with the Dask version may exist.

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** [str] The offset string or object representing target conversion.
- **how** [str (Not supported in Dask)] Method for down/re-sampling, default to ‘mean’ for
downsampling.
  
  Deprecated since version 0.18.0: The new syntax is .resample(...).mean(),
or .resample(...).apply(<func>)
- **axis** [(0 or 'index', 1 or 'columns'), default 0 (Not supported in Dask)] Which axis to
  use for up- or down-sampling. For Series this will default to 0, i.e. along the rows.
  Must be DatetimeIndex, TimedeltaIndex or PeriodIndex.
- **fill_method** [str, default None (Not supported in Dask)] Filling method for upsampling.
  
  Deprecated since version 0.18.0: The new syntax is .resample(...).
  <func>(), e.g .resample(...).pad()
- **closed** [('right', 'left'), default None] Which side of bin interval is closed. The default
  which all have a default of ‘right’.
- **label** [('right', 'left')], default None] Which bin edge label to label bucket with. The
  and ‘W’ which all have a default of ‘right’.
- **convention** [('start', 'end', 's', 'e'), default ‘start’ (Not supported in Dask)] For Peri-
odIndex only, controls whether to use the start or end of rule.
- **kind** [('timestamp', 'period'), optional, default None (Not supported in Dask)] Pass
  ‘timestamp’ to convert the resulting index to a DateTimeIndex or ‘period’ to convert
  it to a PeriodIndex. By default the input representation is retained.
- **loffset** [timedelta, default None (Not supported in Dask)] Adjust the resampled time la-
bels.
- **limit** [int, default None (Not supported in Dask)] Maximum size gap when reindexing
  with fill_method.
  
  Deprecated since version 0.18.0.
- **base** [int, default 0 (Not supported in Dask)] 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** [str, optional (Not supported in Dask)] For a DataFrame, column to use instead of
  index for resampling. Column must be datetime-like.
  
  New in version 0.19.0.
- **level** [str or int, optional (Not supported in Dask)] For a MultiIndex, level (name or num-
  ber) to use for resampling. level must be datetime-like.
  
  New in version 0.19.0.

Returns
Resampler object

See also:

groupby  Group by mapping, function, label, or list of labels.
Series.resample  Resample a Series.
DataFrame.resample  Resample a DataFrame.

Notes

See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```python
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')  # doctest: +SKIP
>>> series = pd.Series(range(9), index=index)  # doctest: +SKIP
>>> series  # doctest: +SKIP
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.

```python
>>> series.resample('3T').sum()  # doctest: +SKIP
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.

```python
>>> series.resample('3T', label='right').sum()  # doctest: +SKIP
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.
Upsample the series into 30 second bins.

```python
>>> series.resample('30S').asfreq()[0:5]  # Select first 5 rows  # doctest: +SKIP
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.

```python
>>> series.resample('30S').pad()[0:5]  # doctest: +SKIP
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
```

Upsample the series into 30 second bins and fill the NaN values using the `bfill` method.

```python
>>> series.resample('30S').bfill()[0:5]  # doctest: +SKIP
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 0
2000-01-01 00:02:00 2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```python
>>> def custom_resampler(array_like):
...     return np.sum(array_like) + 5
...
```

```python
>>> series.resample('3T').apply(custom_resampler)  # doctest: +SKIP
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`.

Resample a year by quarter using `start` `convention`. Values are assigned to the first quarter of the period.

```python
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01', freq='A',
...                index=rule=4,  # doctest: +SKIP
...                freq='A',
...                )
```

(continues on next page)
Resample quarters by month using ‘end’ convention. Values are assigned to the last month of the period.

```python
g = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01', freq='Q', periods=4))
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```python
d = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
          'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
df = pd.DataFrame(d)
df['week_starting'] = pd.date_range('01/01/2018', periods=8, freq='W')
```
For a DataFrame with MultiIndex, the keyword _level_ can be used to specify on which level the resampling needs to take place.

```python
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')  # doctest: +SKIP
>>> d2 = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],  # doctest: +SKIP
...            'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df2 = pd.DataFrame(d2,  # doctest: +SKIP
...                     index=pd.MultiIndex.from_product([[days,  # doctest: +SKIP
...                                                      ...                             ...
...                                                      ...                             ...
...                                                      ...                             ]
...                                                      ]
...                                                      )
...                                                      )
>>> df2  # doctest: +SKIP
       price  volume
2000-01-01 morning 10   50
       afternoon 11   60
2000-01-02 morning  9   40
       afternoon 13  100
2000-01-03 morning 14   50
       afternoon 18  100
2000-01-04 morning 17   40
       afternoon 19   50
```

```python
>>> df2.resample('D', level=0).sum()  # doctest: +SKIP
       price  volume
2000-01-01  21  110
2000-01-02  22  140
2000-01-03  32  150
2000-01-04  36   90
```

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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is 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

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```

`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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns
result [Series]

See also:

Series.mod

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest:+SKIP
>>> a
# doctest: +SKIP
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest:+SKIP
>>> b
# doctest: +SKIP
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```

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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
level [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

Returns
result [Series]

See also:
Series.mul

Examples

```python
d>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
  a  1.0
  b  1.0
  c  1.0
  d  NaN
dtype: float64
d>>> a  # doctest: +SKIP
  a  1.0
  b  1.0
  c  1.0
  d  NaN
dtype: float64
d>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
  a  1.0
  b  NaN
  d  1.0
  e  NaN
dtype: float64
d>>> a.add(b, fill_value=0)  # doctest: +SKIP
  a  2.0
  b  1.0
  c  1.0
  d  1.0
  e  NaN
dtype: float64
```

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.

This docstring was copied from pandas.core.series.Series.round.

Some inconsistencies with the Dask version may exist.

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 existing missing (NaN) values,
and any new element needed for successful Series alignment, with this value before
computation. If data in both corresponding Series locations is missing the result will
be missing

level [int or name] Broadcast across a level, matching Index values on the passed Multi-
Index level

Returns

result [Series]

See also:

Series.pow

Examples

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a  # doctest: +SKIP
a    1.0
b    1.0
c    1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b  # doctest: +SKIP
a    1.0
b    NaN
d    1.0
```

(continues on next page)
```python
e NaN
dtype: float64

>>> a.add(b, fill_value=0)  # doctest: +SKIP
a  2.0
b  1.0
c  1.0
d  1.0
e  NaN
dtype: float64
```

**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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
- **level** [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

- **result** [Series]

See also:

- `Series.sub`

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a  # doctest: +SKIP
a  1.0
b  1.0
c  1.0
d  NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b  # doctest: +SKIP
a  1.0
b  NaN
d  1.0
e  NaN
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a  2.0
b  1.0
c  1.0
```

(continues on next page)
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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing
- `level` [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

- `result` [Series]

**See also:**

- `Series.truediv`

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest:
˓→ SKIP
>>> a  # doctest: +SKIP
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest:
˓→ SKIP
>>> b  # doctest: +SKIP
a 1.0
b NaN
d 1.0
e 1.0
dtype: float64
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```

**sample** 

`(n=None, frac=None, replace=False, random_state=None)`

Random sample of items
Parameters

**n** [int, optional] Number of items to return is not supported by dask. Use frac instead.

**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.

This docstring was copied from pandas.core.frame.DataFrame.sem.

Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

**numeric_only** [boolean, default None (Not supported in Dask)] 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)]

```
shape
```

Return a tuple representing the dimensionality of a Series.

The single element of the tuple is a Delayed result.

Examples

```
>>> series.shape  # doctest: +SKIP
# (dd.Scalar<size-ag..., dtype=int64>,)
```

```
shift(periods=1, freq=None, axis=0)
```

Shift index by desired number of periods with an optional time freq.

This docstring was copied from pandas.core.frame.DataFrame.shift.

Some inconsistencies with the Dask version may exist.

When freq is not passed, shift the index without realigning the data. If freq is passed (in this case, the index must be date or datetime, or it will raise a *NotImplementedError*), the index will be increased using the periods and the freq.
Parameters

periods [int] Number of periods to shift. Can be positive or negative.

freq [DateOffset, tsseries.offsets, timedelta, or str, optional] Offset to use from the tsseries module or time rule (e.g. ‘EOM’). 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.

axis [[0 or ‘index’, 1 or ‘columns’, None], default None] Shift direction.

fill_value [object, optional (Not supported in Dask)] The scalar value to use for newly introduced missing values. the default depends on the dtype of self. For numeric data, np.nan is used. For datetime, timedelta, or period data, etc. NaT is used. For extension dtypes, self.dtype.na_value is used.

Changed in version 0.24.0.

Returns

DataFrame Copy of input object, shifted.

See also:

Index.shift Shift values of Index.

DatetimeIndex.shift Shift values of DatetimeIndex.

PeriodIndex.shift Shift values of PeriodIndex.

tshift Shift the time index, using the index’s frequency if available.

Examples

```python
>>> df = pd.DataFrame({'Col1': [10, 20, 15, 30, 45],  # doctest: +SKIP
                    'Col2': [13, 23, 18, 33, 48],
                    'Col3': [17, 27, 22, 37, 52]})
```

```python
>>> df.shift(periods=3)  # doctest: +SKIP
   Col1  Col2  Col3
0   NaN   NaN   NaN
1   NaN   NaN   NaN
2   NaN   NaN   NaN
3  10.0  13.0  17.0
4  20.0  23.0  27.0
```

```python
>>> df.shift(periods=1, axis='columns')  # doctest: +SKIP
   Col1  Col2  Col3
0   NaN  10.0  13.0
1   NaN  20.0  23.0
2   NaN  15.0  18.0
3   NaN  30.0  33.0
4   NaN  45.0  48.0
```

```python
>>> df.shift(periods=3, fill_value=0)  # doctest: +SKIP
   Col1  Col2  Col3
0    0    0    0
1    0    0    0
2    0    0    0
```

(continues on next page)
size
Size of the Series or DataFrame as a Delayed object.

Examples

```python
>>> series.size  # doctest: +SKIP
dd.Scalar<size-ag..., dtype=int64>
```

squeeze()
Squeeze 1 dimensional axis objects into scalars.

This docstring was copied from pandas.core.series.Series.squeeze.

Some inconsistencies with the Dask version may exist.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don’t know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call `squeeze` to ensure you have a Series.

Parameters

`axis` {0 or ‘index’, 1 or ‘columns’, None}, default None (Not supported in Dask)
A specific axis to squeeze. By default, all length-1 axes are squeezed.

New in version 0.20.0.

Returns

DataFrame, Series, or scalar
The projection after squeezing `axis` or all the axes.

See also:

Series.iloc Integer-location based indexing for selecting scalars.

DataFrame.iloc Integer-location based indexing for selecting Series.

Series.to_frame Inverse of DataFrame.squeeze for a single-column DataFrame.

Examples

```python
>>> primes = pd.Series([2, 3, 5, 7])  # doctest: +SKIP
```

Slicing might produce a Series with a single value:

```python
>>> even_primes = primes[primes % 2 == 0]  # doctest: +SKIP
>>> even_primes  # doctest: +SKIP
0    2
dtype: int64
```

```python
>>> even_primes.squeeze()  # doctest: +SKIP
2
```

Squeezing objects with more than one value in every axis does nothing:
Squeezing is even more effective when used with DataFrames.

Slicing a single column will produce a DataFrame with the columns having only one value:

Squeezing the rows produces a single scalar Series:

Squeezing all axes will project directly into a scalar:

std

Return sample standard deviation over requested axis.

This docstring was copied from pandas.core.frame.DataFrame.std.
Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series
- **ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
- **numeric_only** [boolean, default None (Not supported in Dask)] 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)]

**str**

Namespace for string methods

**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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is 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`

**Examples**

```python
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest:+SKIP
>>> a  # doctest: +SKIP
a    1.0
b    1.0
c    1.0
d  NaN
```

(continues on next page)
dtype: float64

```python
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b  # doctest: +SKIP
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
```

```python
>>> a.add(b, fill_value=0)  # doctest: +SKIP
a   2.0
b   1.0
c   1.0
d   1.0
e   NaN
dtype: float64
```

`sum(axis=None, skipna=True, split_every=False, dtype=None, out=None, min_count=None)`

Return the sum of the values for the requested axis.

This docstring was copied from pandas.core.frame.DataFrame.sum.

Some inconsistencies with the Dask version may exist.

This is equivalent to the method `numpy.sum`.

**Parameters**

- **axis** ([index (0), columns (1)]) Axis for the function to be applied on.
- **skipna** [bool, default True] Exclude NA/null values when computing the result.
- **level** [int or level name, default None (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** [bool, default None (Not supported in Dask)] 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 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

**kwargs Additional keyword arguments to be passed to the function.

**Returns**

- **sum** [Series or DataFrame (if level specified)]

See also:

- **Series.sum** Return the sum.
- **Series.min** Return the minimum.
- **Series.max** Return the maximum.
- **Series.idxmin** Return the index of the minimum.
- **Series.idxmax** Return the index of the maximum.
DataFrame.min  Return the sum over the requested axis.
DataFrame.min  Return the minimum over the requested axis.
DataFrame.max  Return the maximum over the requested axis.
DataFrame.idxmin Return the index of the minimum over the requested axis.
DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```python
>>> idx = pd.MultiIndex.from_arrays([['warm', 'warm', 'cold', 'cold'], ['dog', 'falcon', 'fish', 'spider'], names=['blooded', 'animal'])

>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)

>>> s
blooded animal
   warm  dog  4
       falcon  2
   cold  fish  0
       spider  8
Name: legs, dtype: int64

>>> s.sum() # doctest: +SKIP
14

Sum using level names, as well as indices.

```python
>>> s.sum(level='blooded') # doctest: +SKIP
blooded
   warm  6
   cold  8
Name: legs, dtype: int64
```

```python
>>> s.sum(level=0) # doctest: +SKIP
blooded
   warm  6
   cold  8
Name: legs, dtype: int64
```

By default, the sum of an empty or all-NA Series is 0.

```python
>>> pd.Series([]).sum() # min_count=0 is the default # doctest: +SKIP
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.

```python
>>> pd.Series([], min_count=1) # doctest: +SKIP
nan
```

Thanks to the skipna parameter, min_count handles all-NA and empty series identically.

```python
>>> pd.Series([np.nan]).sum() # doctest: +SKIP
0.0
```
>>> pd.Series([np.nan]).sum(min_count=1)  # doctest: +SKIP
nan

tail \((n=5, \text{compute}=\text{True})\)
Last \(n\) rows of the dataset
Caveat, the only checks the last \(n\) rows of the last partition.

to_bag \((\text{index}=\text{False})\)
Create a Dask Bag from a Series

to_csv \((\text{filename}, **\text{kwargs})\)
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:

```python
>>> 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.

```python
>>> 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)  # doctest: +SKIP
```

```
/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:

```python
>>> 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

header_first_partition_only [boolean, default False] If set, only write the header row in the first output file

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_dask_array** (*lengths=None*)  
Convert a dask DataFrame to a dask array.

**Parameters**

- **lengths** [bool or Sequence of ints, optional] How to determine the chunks sizes for the output array. By default, the output array will have unknown chunk lengths along the first axis, which can cause some later operations to fail.
  
  - True: immediately compute the length of each partition
  - Sequence: a sequence of integers to use for the chunk sizes on the first axis. These values are not validated for correctness, beyond ensuring that the number of items matches the number of partitions.

**to_delayed** (*optimize_graph=True*)  
Convert into a list of dask.delayed objects, one per partition.

**Parameters**

- **optimize_graph** [bool, optional] If True [default], the graph is optimized before converting into dask.delayed objects.

**See also:**

dask.dataframe.from_delayed

**Examples**

```python
>>> partitions = df.to_delayed()  # doctest: +SKIP
```

**to_frame** (*name=None*)  
Convert Series to DataFrame.

This docstring was copied from pandas.core.series.Series.to_frame. Some inconsistencies with the Dask version may exist.

**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, **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 asterix * within the filename or datapath, and an optional name_function. The asterix 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.

**scheduler** [string] The scheduler to use, like “threads” or “processes”

**other:** See pandas.to_hdf for more information

**Returns**

**filenames** [list] Returned if `compute` is True. List of file names that each partition is saved to.

**delayed** [dask.Delayed] Returned if `compute` is False. Delayed object to execute to_hdf when computed.

**See also:**

`read_hdf`, `to_parquet`

**Examples**

Save Data to a single file

```python
>>> df.to_hdf('output.hdf', '/data')  # doctest: +SKIP
```

Save data to multiple datapaths within the same file:

```python
>>> df.to_hdf('output.hdf', '/data-*')  # doctest: +SKIP
```

Save data to multiple files:

```python
>>> df.to_hdf('output-*hdf', '/data')  # doctest: +SKIP
```

Save data to multiple files, using the multiprocessing scheduler:

```python
>>> df.to_hdf('output-*hdf', '/data', scheduler='processes')  # doctest: +SKIP
```


```python
>>> 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)
```
```python
>>> df.to_hdf('*.hdf', '/data', name_function=name_function) # doctest: +SKIP
```

```python
to_json(filename, *args, **kwargs)
```
See dd.to_json docstring for more information

```python
to_string(max_rows=5)
```
Render a string representation of the Series.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **buf** [StringIO-like, optional (Not supported in Dask)] buffer to write to
- **na_rep** [string, optional (Not supported in Dask)] string representation of NaN to use, default ‘NaN’
- **float_format** [one-parameter function, optional (Not supported in Dask)] formatter function to apply to columns’ elements if they are floats default None
- **header** [boolean, default True (Not supported in Dask)] Add the Series header (index name)
- **index** [bool, optional (Not supported in Dask)] Add index (row) labels, default True
- **length** [boolean, default False (Not supported in Dask)] Add the Series length
- **dtype** [boolean, default False (Not supported in Dask)] Add the Series dtype
- **name** [boolean, default False (Not supported in Dask)] 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)]

```python
to_timestamp(freq=None, how='start', axis=0)
```
Cast to DatetimeIndex of timestamps, at beginning of period.

Some inconsistencies with the Dask version may exist.

**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 (Not supported in Dask)] If false then underlying input data is not copied

**Returns**

- **df** [DataFrame with DatetimeIndex]

```python
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 existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

- `level` [int or name] Broadcast across a level, matching Index values on the passed Multi-Index level

**Returns**

- `result` [Series]

**See also:**

* `Series.rtruediv`

**Examples**

```python
d>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])  # doctest: +SKIP
>>> a
a 1.0
b 1.0
c 1.0
d NaN
dtype: float64
```

```python
d>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])  # doctest: +SKIP
>>> b
a 1.0
b NaN
d 1.0
e NaN
dtype: float64
```

```python
d>>> a.add(b, fill_value=0)  # doctest: +SKIP
a 2.0
b 1.0
c 1.0
d 1.0
e NaN
dtype: float64
```

**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)*

Return a Series containing counts of unique values.

This docstring was copied from pandas.core.series.Series.value_counts.
Some inconsistencies with the Dask version may exist.

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 (Not supported in Dask)] If True then the object returned will contain the relative frequencies of the unique values.
- **sort** [boolean, default True (Not supported in Dask)] Sort by values.
- **ascending** [boolean, default False (Not supported in Dask)] Sort in ascending order.
- **bins** [integer, optional (Not supported in Dask)] Rather than count values, group them into half-open bins, a convenience for `pd.cut`, only works with numeric data.
- **dropna** [boolean, default True (Not supported in Dask)] Don’t include counts of NaN.

**Returns**

- **counts** [Series]

**See also:**

- **Series.count** Number of non-NA elements in a Series.
- **DataFrame.count** Number of non-NA elements in a DataFrame.

**Examples**

```python
>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])  # doctest: +SKIP
>>> index.value_counts()  # doctest: +SKIP
3.0 2
4.0 1
2.0 1
1.0 1
dtype: int64
```

With `normalize` set to `True`, returns the relative frequency by dividing all values by the sum of values.

```python
>>> s = pd.Series([3, 1, 2, 3, 4, np.nan])  # doctest: +SKIP
>>> s.value_counts(normalize=True)  # doctest: +SKIP
3.0 0.4
4.0 0.2
2.0 0.2
1.0 0.2
dtype: float64
```

**bins**

Bins can be useful for going from a continuous variable to a categorical variable; instead of counting unique apparitions of values, divide the index in the specified number of half-open bins.

```python
>>> s.value_counts(bins=3)  # doctest: +SKIP
(2.0, 3.0] 2
(0.996, 2.0] 2
(3.0, 4.0] 1
dtype: int64
```
dropna

With dropna set to False we can also see NaN index values.

```python
>>> s.value_counts(dropna=False)  # doctest: +SKIP
3.0    2
NaN    1
4.0    1
2.0    1
1.0    1
dtype: int64
```

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

Return unbiased variance over requested axis.

This docstring was copied from pandas.core.frame.DataFrame.var.
Some inconsistencies with the Dask version may exist.
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 (Not supported in Dask)] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series
- `ddof` [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
- `numeric_only` [boolean, default None (Not supported in Dask)] 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)]

visualize

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.
**color**: `{None, ‘order’}`, **optional**  Options to color nodes. Provide `cmap=` keyword for additional color map.

**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:

https://docs.dask.org/en/latest/optimize.html

**Examples**

```python
>>> x.visualize(filename='dask.pdf')  # doctest: +SKIP
>>> x.visualize(filename='dask.pdf', color='order')  # doctest: +SKIP
```

**where** *(cond, other=nan)*

Replace values where the condition is False.

This docstring was copied from pandas.core.frame.DataFrame.where.

Some inconsistencies with the Dask version may exist.

**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 (Not supported in Dask)]  Whether to perform the operation in place on the data.

**axis**  [int, default None (Not supported in Dask)]  Alignment axis if needed.

**level**  [int, default None (Not supported in Dask)]  Alignment level if needed.

**errors**  [str, {'raise', 'ignore'}, default raise (Not supported in Dask)]  Note that currently this parameter won’t affect the results and will always coerce to a suitable dtype.

- **raise**: allow exceptions to be raised.
- **ignore**: suppress exceptions. On error return original object.
try_cast [boolean, default False (Not supported in Dask)] Try to cast the result back to the input type (if possible).

raise_on_error [boolean, default True (Not supported in Dask)] Whether to raise on invalid data types (e.g. trying to where on strings).

Deprecated since version 0.21.0: Use errors.

Returns

wh [same type as caller]

See also:

DataFrame.mask() Return an object of same shape as self.

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

```python
>>> s = pd.Series(range(5))  # doctest: +SKIP
>>> s.where(s > 0)  # doctest: +SKIP
0  NaN
1  1.0
2  2.0
3  3.0
4  4.0
dtype: float64
```

```python
>>> s.mask(s > 0)  # doctest: +SKIP
0  0.0
1  NaN
2  NaN
3  NaN
4  NaN
dtype: float64
```

```python
>>> s.where(s > 1, 10)  # doctest: +SKIP
0  10
1  10
2  2
3  3
4  4
dtype: int64
```
```python
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0  # doctest: +SKIP
>>> df.where(m, -df)  # doctest: +SKIP
   A  B
0  0  -1
1  1  -2
2  2  -3
3  3  -4
4  4  -5
5  5  -6
6  6  -7
7  7  -8
8  8  -9
9  9  10
```

DataFrameGroupBy

```python
>>> df.where(m, -df) == np.where(m, df, -df)  # doctest: +SKIP
   A  B
0  True True
1  True True
2  True True
3  True True
4  True True
5  True True
6  True True
7  True True
8  True True
9  True True
```

```python
>>> df.where(m, -df) == df.mask(~m, -df)  # doctest: +SKIP
   A  B
0  True True
1  True True
2  True True
3  True True
4  True True
5  True True
6  True True
7  True True
8  True True
9  True True
```

```python
>>> df.groupby('A').sum()  # doctest: +SKIP
   B
A
-1
```

```python
>>> df.groupby('A').agg('sum')  # doctest: +SKIP
   B
A
-1
```

```python
dataframe.groupby.DataFrameGroupBy

**class** dask.dataframe.groupby.DataFrameGroupBy(df, by=None, slice=None, group_keys=True)

**agg**(arg, split_every=None, split_out=1)

Aggregate using one or more operations over the specified axis.

This docstring was copied from pandas.core.groupby.generic.DataFrameGroupBy.agg.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **func** [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

  Accepted combinations are:
  - function
  - string function name
  - list of functions and/or function names, e.g. [np.sum, 'mean']
  - dict of axis labels -> functions, function names or list of such.

- **args** Positional arguments to pass to func.

- **kwargs** Keyword arguments to pass to func.

**Returns**

- **DataFrame, Series or scalar** if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if
Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

pandas.DataFrame.groupby.apply, pandas.DataFrame.groupby.transform, pandas.DataFrame.aggregate

Notes

agg is an alias for aggregate. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```python
>>> df = pd.DataFrame({'A': [1, 1, 2, 2],  # doctest: +SKIP
...                   'B': [1, 2, 3, 4],
...                   'C': np.random.randn(4)})
```

The aggregation is for each column.

```python
>>> df.groupby('A').agg('min')  # doctest: +SKIP
       B     C
   A
1  1  0.227877  0.362838
2  3 -0.562860  1.267767
```

Multiple aggregations

```python
>>> df.groupby('A').agg(['min', 'max'])  # doctest: +SKIP
                   B     C
    min   max    min   max
   A
1  1  0.227877  0.362838
2  3 -0.562860  1.267767
```

Select a column for aggregation

```python
>>> df.groupby('A').B.agg(['min', 'max'])  # doctest: +SKIP
      min   max
   A
1  1   2
2  3 -4
```

Different aggregations per column

```python
>>> df.groupby('A').B.agg(['min', 'max'])  # doctest: +SKIP
      min   max
   A
1  1   2
2  3 -4
```
aggregate (arg, split_every=None, split_out=1)

Aggregate using one or more operations over the specified axis.

This docstring was copied from pandas.core.groupby.generic.DataFrameGroupBy.aggregate.

Some inconsistencies with the Dask version may exist.

Parameters

- **func** [function, str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.
  
  Accepted combinations are:
  
  - function
  
  - string function name
  
  - list of functions and/or function names, e.g. [np.sum, 'mean']
  
  - dict of axis labels -> functions, function names or list of such.

- **args** Positional arguments to pass to `func`.

- **kwargs** Keyword arguments to pass to `func`.

Returns

- **DataFrame, Series or scalar** if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:

- pandas.DataFrame.groupby.apply
- pandas.DataFrame.groupby.transform
- pandas.DataFrame.aggregate

Notes

- **agg** is an alias for aggregate. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```python
>>> df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4], 'C': np.random.randn(4))
```
>>> df  
# doctest: +SKIP
  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')  
# doctest: +SKIP
  B   C
  A
  1   1  0.227877
  2   3 -0.562860

Multiple aggregations

>>> df.groupby('A').agg(['min', 'max'])  
# doctest: +SKIP
  B   C
  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'])  
# doctest: +SKIP
  min max
  A
  1   1  2
  2   3  4

Different aggregations per column

>>> df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})  
# doctest: +SKIP
  B   C
  min max sum
  A
  1   1  2  0.590716
  2   3  4  0.704907

**apply** *(func, *args, **kwargs)*

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

- **args, kwargs**: [Scalar, Delayed or object]  
  Arguments and keywords to pass to the 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

applied [Series or DataFrame depending on columns keyword]

count (split_every=None, split_out=1)
Compute count of group, excluding missing values.
This docstring was copied from pandas.core.groupby.groupby.GroupBy.count.
Some inconsistencies with the Dask version may exist.

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.
This docstring was copied from pandas.core.groupby.groupby.GroupBy.cumcount.
Some inconsistencies with the Dask version may exist.
Essentially this is equivalent to

```python
>>> self.apply(lambda x: pd.Series(np.arange(len(x)), x.index))  # doctest: +SKIP
```

Parameters

ascending [bool, default True (Not supported in Dask)] If False, number in reverse, from length of group - 1 to 0.

See also:

ngroup Number the groups themselves.

Examples

```python
>>> df = pd.DataFrame([['a'], ['a'], ['a'], ['b'], ['b'], ['a']],
    columns=['A'])
```

```python
>>> df  # doctest: +SKIP
A
0 a
1 a
2 a
3 b
4 b
```
cumprod (axis=0)
Cumulative product for each group.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.cumprod.

Some inconsistencies with the Dask version may exist.

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

cumsum (axis=0)
Cumulative sum for each group.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.cumsum.

Some inconsistencies with the Dask version may exist.

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

first (split_every=None, split_out=1)
Compute first of group values See Also ——– pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

groupby (key)
Constructs NDFrame from group with provided name.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.get_group.

Some inconsistencies with the Dask version may exist.

Parameters

name [object (Not supported in Dask)] the name of the group to get as a DataFrame

obj [NDFrame, default None (Not supported in Dask)] the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

Returns

group [same type as obj]
last (split_every=None, split_out=1)
Compute last of group values

See Also ——–
pandas.Series.groupby
pandas.DataFrame.groupby
pandas.Panel.groupby

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.
This docstring was copied from pandas.core.groupby.groupby.GroupBy.mean.
Some inconsistencies with the Dask version may exist.

Returns
pandas.Series or pandas.DataFrame

```python
>>> B C
A
1  3.0  1.333333
2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```python
>>> df.groupby(['A', 'B']).mean()  # doctest: +SKIP
```

```python
>>> C
A B
1  2.0  2
   4.0  1
2  3.0  1
   5.0  2
```

Groupby one column and return the mean of only particular column in the group.

```python
>>> df.groupby('A')['B'].mean()  # doctest: +SKIP
```

```python
>>> A
1  3.0
2  4.0
Name: B, dtype: float64
```

min (split_every=None, split_out=1)
Compute min of group values

See Also ——–
pandas.Series.groupby
pandas.DataFrame.groupby
pandas.Panel.groupby

prod (split_every=None, split_out=1, min_count=None)
Compute prod of group values

See Also ——–
pandas.Series.groupby
pandas.DataFrame.groupby
pandas.Panel.groupby
size(split_every=None, split_out=1)
Compute group sizes.
This docstring was copied from pandas.core.groupby.groupby.GroupBy.size.
Some inconsistencies with the Dask version may exist.
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.
This docstring was copied from pandas.core.groupby.groupby.GroupBy.std.
Some inconsistencies with the Dask version may exist.
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, min_count=None)
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.
This docstring was copied from pandas.core.groupby.groupby.GroupBy.var.
Some inconsistencies with the Dask version may exist.
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, **kwargs)
agg(arg, split_every=None, split_out=1)
Aggregate using one or more operations over the specified axis.
This docstring was copied from pandas.core.groupby.generic.SeriesGroupBy.agg.
Some inconsistencies with the Dask version may exist.
Parameters
    func [function, str, list or dict] Function to use for aggregating the data. If a function,
    must either work when passed a Series or when passed to Series.apply.
    Accepted combinations are:
• function
• string function name
• list of functions and/or function names, e.g. [np.sum, 'mean']
• dict of axis labels -> functions, function names or list of such.

*args Positional arguments to pass to func.

**kwargs Keyword arguments to pass to func.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns a Series if DataFrame.agg is called with several functions, returns a DataFrame if Series.agg is called with single function, returns a scalar if Series.agg is called with several functions, returns a Series

See also:
pandas.Series.groupby.apply, pandas.Series.groupby.transform, pandas.Series.aggregate

Notes

agg is an alias for aggregate. Use the alias.
A passed user-defined-function will be passed a Series for evaluation.

Examples

>>> s = pd.Series([1, 2, 3, 4])  # doctest: +SKIP

>>> s  # doctest: +SKIP
0  1
1  2
2  3
3  4
dtype: int64

>>> s.groupby([1, 1, 2, 2]).min()  # doctest: +SKIP
1  1
2  3
dtype: int64

>>> s.groupby([1, 1, 2, 2]).agg('min')  # doctest: +SKIP
1 1
2 3
dtype: int64

>>> s.groupby([1, 1, 2, 2]).agg(['min', 'max'])  # doctest: +SKIP
    min  max
1  1   2
2  3   4
aggregate(arg, split_every=None, split_out=1)

Aggregate using one or more operations over the specified axis.

This docstring was copied from pandas.core.groupby.generic.SeriesGroupBy.aggregate.

Some inconsistencies with the Dask version may exist.

Parameters

   func [function, str, list or dict] Function to use for aggregating the data. If a function,
   must either work when passed a Series or when passed to Series.apply.
   Accepted combinations are:
   • function
   • string function name
   • list of functions and/or function names, e.g. [np.sum, 'mean']
   • dict of axis labels -> functions, function names or list of such.

*args Positional arguments to pass to func.

**kwargs Keyword arguments to pass to func.

Returns

DataFrame, Series or scalar if DataFrame.agg is called with a single function, returns
a Series if DataFrame.agg is called with several functions, returns a DataFrame if
Series.agg is called with single function, returns a scalar if Series.agg is called with
several functions, returns a Series

See also:
pandas.Series.groupby.apply, pandas.Series.groupby.transform, pandas.Series.aggregate

Notes

agg is an alias for aggregate. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```python
groupby([1, 1, 2, 2]).min()  # doctest: +SKIP
```

```python
s = pd.Series([1, 2, 3, 4])  # doctest: +SKIP
```

```python
s.groupby([1, 1, 2, 2]).min()  # doctest: +SKIP
```

```python
s = pd.Series([1, 2, 3, 4])  # doctest: +SKIP
```
```python
>>> s.groupby([1, 1, 2, 2]).agg('min')  # doctest: +SKIP
1  1
2  3
dtype: int64
```

```python
>>> s.groupby([1, 1, 2, 2]).agg(['min', 'max'])  # doctest: +SKIP
      min   max
1  1    2
2  3    4
```

**apply** *(func, *args, **kwargs)*

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
- **args, kwargs**  
  [Scalar, Delayed or object] Arguments and keywords to pass to the 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**

- **applied**  
  [Series or DataFrame depending on columns keyword]

**count** *(split_every=None, split_out=1)*

Compute count of group, excluding missing values.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.count.

Some inconsistencies with the Dask version may exist.

**See also**:


**cumcount** *(axis=None)*

Number each item in each group from 0 to the length of that group - 1.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.cumcount.

Some inconsistencies with the Dask version may exist.

Essentially this is equivalent to
```python
>>> self.apply(lambda x: pd.Series(np.arange(len(x)), x.index))  # doctest: +SKIP
```

**Parameters**

- **ascending** [bool, default True (Not supported in Dask)] If False, number in reverse, from length of group - 1 to 0.

**See also:**

- **ngroup** Number the groups themselves.

**Examples**

```python
>>> df = pd.DataFrame([['a'], ['a'], ['a'], ['b'], ['b'], ['a']],
                    columns=['A'])
```

```python
>>> df  # doctest: +SKIP
    A
0  a
1  a
2  a
3  b
4  b
5  a
```

```python
>>> df.groupby('A').cumcount()  # doctest: +SKIP
0  0
1  1
2  2
3  0
4  1
5  3
dtype: int64
```

```python
>>> df.groupby('A').cumcount(ascending=False)  # doctest: +SKIP
0  3
1  2
2  1
3  1
4  0
5  0
```

cumprod (axis=0)

Cumulative product for each group.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.cumprod.

Some inconsistencies with the Dask version may exist.

**See also:**

- pandas.Series.groupby
- pandas.DataFrame.groupby
- pandas.Panel.groupby

cumsum (axis=0)

Cumulative sum for each group.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.cumsum.
Some inconsistencies with the Dask version may exist.

See also:

pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

**first** *(split_every=None, split_out=1)*

Compute first of group values

See Also —— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

**get_group** *(key)*

Constructs NDFrame from group with provided name.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.get_group.

Some inconsistencies with the Dask version may exist.

**Parameters**

- **name** [object (Not supported in Dask)] the name of the group to get as a DataFrame
- **obj** [NDFrame, default None (Not supported in Dask)] the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

**Returns**

- **group** [same type as obj]

**last** *(split_every=None, split_out=1)*

Compute last of group values

See Also —— pandas.Series.groupby pandas.DataFrame.groupby pandas.Panel.groupby

**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.

This docstring was copied from pandas.core.groupby.groupby.GroupBy.mean.

Some inconsistencies with the Dask version may exist.

**Returns**

pandas.Series or pandas.DataFrame

```
>>> B   C
  A
 1  3.0  1.333333
 2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```python
>>> df.groupby(['A', 'B']).mean()  # doctest: +SKIP
```

```python
>>> C
  A   B
  A B
```

(continues on next page)
Groupby one column and return the mean of only particular column in the group.

```python
>>> df.groupby('A')['B'].mean()  # doctest: +SKIP

A
1 3.0
2 4.0
Name: B, dtype: float64
```

### min

This docstring was copied from pandas.core.groupby.groupby.GroupBy.size.
Some inconsistencies with the Dask version may exist.

#### Parameters

- `ddof` [integer, default 1] degrees of freedom

### std

This docstring was copied from pandas.core.groupby.groupby.GroupBy.std.
Some inconsistencies with the Dask version may exist.

#### Parameters

- `ddof` [integer, default 1] degrees of freedom

### sum

This docstring was copied from pandas.core.groupby.groupby.GroupBy.var.
Some inconsistencies with the Dask version may exist.
For multiple groupings, the result index will be a MultiIndex.

Parameters

ddf [integer, default 1] degrees of freedom

See also:
pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby

Custom Aggregation

class dask.dataframe.groupby.Aggregation(name, chunk, agg, finalize=None)

User defined groupby-aggregation.

This class allows users to define their own custom aggregation in terms of operations on Pandas dataframes in a map-reduce style. You need to specify what operation to do on each chunk of data, how to combine those chunks of data together, and then how to finalize the result.

Parameters

name [str] the name of the aggregation. It should be unique, since intermediate result will be identified by this name.

chunk [callable] a function that will be called with the grouped column of each partition. It can either return a single series or a tuple of series. The index has to be equal to the groups.

agg [callable] a function that will be called to aggregate the results of each chunk. Again the argument(s) will be grouped series. If chunk returned a tuple, agg will be called with all of them as individual positional arguments.

finalize [callable] an optional finalizer that will be called with the results from the aggregation.

Examples

We could implement sum as follows:

```python
>>> custom_sum = dd.Aggregation(
...     name='custom_sum',
...     chunk=lambda s: s.sum(),
...     agg=lambda s0: s0.sum()
... ) # doctest: +SKIP
>>> df.groupby('g').agg(custom_sum) # doctest: +SKIP
```

We can implement mean as follows:

```python
>>> custom_mean = dd.Aggregation(
...     name='custom_mean',
...     chunk=lambda s: (s.count(), s.sum()),
...     agg=lambda count, sum: (count.sum(), sum.sum()),
...     finalize=lambda count, sum: sum / count,
... ) # doctest: +SKIP
>>> df.groupby('g').agg(custom_mean) # doctest: +SKIP
```

Though of course, both of these are built-in and so you don’t need to implement them yourself.
Storage and Conversion

dask.dataframe.read_csv(urlpath, blocksize=64000000, collection=True, lineterminator=None, compression=None, enforce=False, assume_missing=False, storage_options=None, include_path_column=False, **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')  # doctest: +SKIP

- In some cases it can break up large files:

  >>> df = dd.read_csv('largefile.csv', blocksize=25e6)  # 25MB chunks  #
  \→ doctest: +SKIP

- It can read CSV files from external resources (e.g. S3, HDFS) by providing a URL:

  >>> df = dd.read_csv('s3://bucket/myfiles.*.csv')  # doctest: +SKIP
  >>> df = dd.read_csv('hdfs://myfiles.*.csv')  # doctest: +SKIP
  >>> df = dd.read_csv('hdfs://namenode.example.com/myfiles.*.csv')  #
  \→ doctest: +SKIP

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 or list] Absolute or relative filepath(s). Prefix with a protocol like s3:// to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol.

blocksize [str, 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. Can be a number like 64000000 or a string like “64MB”

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.

include_path_column [bool or str, optional] Whether or not to include the path to each particular file. If True a new column is added to the dataframe called path. If str, sets new column name. Default is False.

**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, include_path_column=False, **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:

```python
>>> df = dd.read_table('myfiles.*.csv')  # doctest: +SKIP
```

- In some cases it can break up large files:

```python
>>> 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:

```python
>>> df = dd.read_table('s3://bucket/myfiles.*.csv')  # doctest: +SKIP
>>> df = dd.read_table('hdfs://myfiles.*.csv')  # doctest: +SKIP
>>> 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 or list] Absolute or relative filepath(s). Prefix with a protocol like s3:// to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol.
- **blocksize** [str, 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. Can be a number like 64000000 or a string like “64MB”
- **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.

**include_path_column** [bool or str, optional] Whether or not to include the path to each particular file. If True a new column is added to the dataframe called `path`. If str, sets new column name. Default is False.

**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_fwf(urlpath, blocksize=64000000, collection=True, lineterminator=None, compression=None, sample=256000, enforce=False, assume_missing=False, storage_options=None, include_path_column=False, **kwargs)

Read fixed-width files into a Dask.DataFrame

This parallelizes the `pandas.read_fwf()` function in the following ways:

- It supports loading many files at once using globstrings:

  ```python
  >>> df = dd.read_fwf('myfiles.*.csv')  # doctest: +SKIP
  ```

- In some cases it can break up large files:

  ```python
  >>> df = dd.read_fwf('largefile.csv', blocksize=25e6)  # 25MB chunks  #
  ```

- It can read CSV files from external resources (e.g. S3, HDFS) by providing a URL:

  ```python
  >>> df = dd.read_fwf('s3://bucket/myfiles.*.csv')  # doctest: +SKIP
  >>> df = dd.read_fwf('hdfs://myfiles.*.csv')  # doctest: +SKIP
  >>> df = dd.read_fwf('hdfs://namenode.example.com/myfiles.*.csv')  #
  ```

Internally `dd.read_fwf` uses `pandas.read_fwf()` and supports many of the same keyword arguments with the same performance guarantees. See the docstring for `pandas.read_fwf()` for more information on available keyword arguments.

**Parameters**
**urlpath** [string or list] Absolute or relative filepath(s). Prefix with a protocol like `s3://` to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol.

**blocksize** [str, 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. Can be a number like `64000000` or a string like “64MB”

**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.

**include_path_column** [bool or str, optional] Whether or not to include the path to each particular file. If True a new column is added to the dataframe called `path`. If str, sets new column name. Default is False.

**kwargs** Extra keyword arguments to forward to `pandas.read_fwf()`.

**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 fixed-width 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', infer_divisions=None)

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, list or fastparquet.ParquetFile] Source directory for data, or path(s) to individual parquet files. Prefix with a protocol like `s3://` to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol. Alternatively, also accepts a previously opened `fastparquet.ParquetFile()`
columns [string, list or None (default)] Field name(s) to read in as columns in the output. By default all non-index fields will be read (as determined by the pandas parquet metadata, if present). Provide a single field name instead of a list to read in the data as a Series.

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, list, False or None (default)] Field name(s) to use as the output frame index. By default will be inferred from the pandas parquet file metadata (if present). Use False to read all fields as columns.

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.

inge [‘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’

infer_divisions [bool or None (default)].] By default, divisions are inferred if the read engine supports doing so efficiently and the index of the underlying dataset is sorted across the individual parquet files. Set to True to force divisions to be inferred in all cases. Note that this may require reading metadata from each file in the dataset, which may be expensive. Set to False to never infer divisions.

See also:

to_parquet

Examples

```python
>>> df = dd.read_parquet('s3://bucket/my-parquet-data')  # doctest: +SKIP
```

dask.dataframe.read_orc (path, columns=None, storage_options=None)

Read dataframe from ORC file(s)

Parameters

- path: str or list(str) Location of file(s), which can be a full URL with protocol specifier, and may include glob character if a single string.

- columns: None or list(str) Columns to load. If None, loads all.

- storage_options: None or dict Further parameters to pass to the bytes backend.

Returns

Dask.DataFrame (even if there is only one column)

Examples

```python
>>> df = dd.read_orc('https://github.com/apache/orc/raw/
... 'master/examples/demo-11-zlib.orc')  # doctest: +SKIP
```
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

```python
>>> dd.read_hdf('myfile.1.hdf5', '/x') # doctest: +SKIP
```

Load multiple files

```python
>>> dd.read_hdf('myfile.*.hdf5', '/x') # doctest: +SKIP
```

```python
>>> dd.read_hdf(['myfile.1.hdf5', 'myfile.2.hdf5'], '/x') # doctest: +SKIP
```

Load multiple datasets

```python
>>> dd.read_hdf('myfile.1.hdf5', '/*') # doctest: +SKIP
```

dask.dataframe.read_json(url_path, orient='records', lines=None, storage_options=None, blocksize=None, sample=1048576, encoding='utf-8', errors='strict', compression='infer', meta=None, **kwargs)

Create a dataframe from a set of JSON files
This utilises pandas.read_json(), and most parameters are passed through - see its docstring.

Differences: orient is 'records' by default, with lines=True; this is appropriate for line-delimited “JSON-lines” data, the kind of JSON output that is most common in big-data scenarios, and which can be chunked when reading (see read_json()). All other options require blocksize=None, i.e., one partition per input file.

Parameters

- **url_path**: str, list of str  
  Location to read from. If a string, can include a glob character to find a set of file names. Supports protocol specifications such as "s3://".

- **encoding, errors**: The text encoding to implement, e.g., “utf-8” and how to respond to errors in the conversion (see str.encode()).

- **orient, lines, kwargs** passed to pandas; if not specified, lines=True when orient='records', False otherwise.

- **storage_options**: dict  
  Passed to backend file-system implementation

- **blocksize**: None or int  
  If None, files are not blocked, and you get one partition per input file. If int, which can only be used for line-delimited JSON files, each partition will be approximately this size in bytes, to the nearest newline character.

- **sample**: int  
  Number of bytes to pre-load, to provide an empty dataframe structure to any blocks without data. Only relevant is using blocksize.

- **encoding, errors**: Text conversion, see bytes.decode()

- **compression** [string or None] String like ‘gzip’ or ‘xz’.

- **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
dask.DataFrame

Examples

Load single file

```python
>>> dd.read_json('myfile.1.json')  # doctest: +SKIP
```

Load multiple files

```python
>>> dd.read_json('myfile.*.json')  # doctest: +SKIP
```

```python
>>> dd.read_json(['myfile.1.json', 'myfile.2.json'])  # doctest: +SKIP
```

Load large line-delimited JSON files using partitions of approx 256MB size

```python
>> dd.read_json('data/file*.csv', blocksize=2**28)
```
dask.dataframe.read_sql_table(table, uri, index_col=None, divisions=None, npartitions=None, limits=None, columns=None, bytes_per_chunk=268435456, head_rows=5, schema=None, meta=None, engine_kwargs=None, **kwargs)

Create dataframe from an SQL table.

If neither divisions or npartitions is given, the memory footprint of the first few rows will be determined, and partitions of size ~256MB will be used.

Parameters

- **table** [string or sqlalchmey expression] Select columns from here.
- **uri** [string] Full sqlalchmey 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 any orderable type. If the type is number or time, then partition boundaries can be inferred from npartitions or bytes_per_chunk; otherwise must supply explicit divisions. index_col 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** Values of the index column to split the table by. If given, this will override npartitions and bytes_per_chunk. The divisions are the value boundaries of the index column used to define the partitions. For example, divisions=list('acegikmoqsuwz') could be used to partition a string column lexicographically into 12 partitions, with the implicit assumption that each partition contains similar numbers of records.
- **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. The index column must be numeric or time for this to work
- **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 sqlalchmey 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
- **head_rows** [int] How many rows to load for inferring the data-types, unless passing meta
- **meta** [empty DataFrame or None] If provided, do not attempt to infer dtypes, but use these, coercing all chunks on load
- **schema** [str or None] If using a table name, pass this to sqlalchmey to select which DB schema to use within the URI connection
- **engine_kwargs** [dict or None] Specific db engine parameters for sqlalchmey
- **kwargs** [dict] Additional parameters to pass to pd.read_sql()
Examples

```python
>>> df = dd.read_sql_table('accounts', 'sqlite:///path/to/bank.db',
                         npartitions=10, index_col='id')  # doctest: +SKIP
```

**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**

- **data** [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

---

4.9. DataFrame
Examples

```python
>>> 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
# doctest: +NORMALIZE_WHITESPACE
(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
# doctest: +NORMALIZE_WHITESPACE
(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/](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, index=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
  - `index` [dask.dataframe.Index, optional] An optional dask Index to use for the output Series or DataFrame.

The default output index depends on whether `x` has any unknown chunks. If there are any unknown chunks, the output has None for all the divisions (one per chunk). If all the chunks are known, a default index with known divisions is created.

Specifying `index` can be useful if you’re conforming a Dask Array to an existing dask Series or DataFrame, and you would like the indices to match.

**See also:**
dask.bag.to_dataframe from dask.bag

dask.dataframe._Frame.values Reverse conversion

dask.dataframe._Frame.to_records Reverse conversion

Examples

```python
>>> 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 https://docs.dask.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_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

```python
>>> df.to_records()  # doctest: +SKIP
dask.array<shape=(nan,), dtype=(numpy.record, [('ind', '<f8'), ('x', '<f8'), ('y', '<i8')]), chunksize=(nan,)>
```
dask.dataframe.to_csv(df, filename, name_function=None, compression=None, compute=True, scheduler=None, storage_options=None, header_first_partition_only=False, **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:

```python
>>> 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.

```python
>>> 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'
```

```python
>>> df.to_csv('/path/to/data/export-*.csv', name_function=name)  # doctest: +SKIP
```

```
/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:

```python
>>> 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

header_first_partition_only [boolean, default False] If set, only write the header row in the first output file

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, optional] If True, the elements are tuples of (index, value), otherwise they're just the value. Default is False.
Examples

```python
>>> bag = df.to_bag()  # doctest: +SKIP
```

dask.dataframe.to_hdf(df, path, key, mode='a', append=False, scheduler=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 asterix 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.
- `scheduler` [string] The scheduler to use, like “threads” or “processes”

**other:** See pandas.to_hdf for more information

**Returns**

- `filenames` [list] Returned if `compute` is True. List of file names that each partition is saved to.
- `delayed` [dask.Delayed] Returned if `compute` is False. Delayed object to execute to_hdf when computed.

See also:

```
read_hdf, to_parquet
```

Examples

Save Data to a single file

```python
>>> df.to_hdf('output.hdf', '/data')  # doctest: +SKIP
```

Save data to multiple datapaths within the same file:
Save data to multiple files:

```python
>>> df.to_hdf('output.hdf', '/data-*')  # doctest: +SKIP
```

Save data to multiple files, using the multiprocessing scheduler:

```python
>>> df.to_hdf('output-*.hdf', '/data', scheduler='processes') # doctest: +SKIP
```


```python
>>> 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) # doctest: +SKIP
```

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]
  - [dask.dataframe.DataFrame]

- **path** [string] Destination directory for data. Prepended 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**

```python
>>> df = dd.read_csv(...)  # doctest: +SKIP
>>> dd.to_parquet(df, '/path/to/output/', compression='snappy')  # doctest: +SKIP
```

dask.dataframe.to_json(df, url_path, orient='records', lines=None, storage_options=None, **kwargs)

Write dataframe into JSON text files

This utilises pandas.DataFrame.to_json(), and most parameters are passed through - see its docstring.

Differences: orient is ‘records’ by default, with lines=True; this produces the kind of JSON output that is most common in big-data applications, and which can be chunked when reading (see `read_json()`).

**Parameters**

- **df**: dask.DataFrame Data to save
- **url_path**: str, list of str Location to write to. If a string, and there are more than one partitions in df, should include a glob character to expand into a set of file names, or provide a name_function= parameter. Supports protocol specifications such as "s3://".
- **orient, lines, kwargs** passed to pandas; if not specified, lines=True when orient='records', False otherwise.
- **storage_options**: dict Passed to backend file-system implementation
- **compute**: bool If true, immediately executes. If False, returns a set of delayed objects, which can be computed at a later time.
- **encoding, errors**: Text conversion, see str.encode()
- **compression**: [string or None] String like ‘gzip’ or ‘xz’.

**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.

scheduler [string, optional] Which scheduler to use like “threads”, “synchronous” or “processes”. 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 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 function.

Examples

```python
>>> 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:

```python
>>> compute({'a': a, 'b': b, 'c': 1})  # doctest: +SKIP
({'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. Arguments and keywords may contain Scalar, Delayed
or regular python objects. DataFrame-like args (both dask and pandas) will be reparti-

tioned to align (if necessary) before applying the function.


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.

```python
>>> a
dd.DataFrame<x, divisions=(1, 3, 5)> # doctest: +SKIP
>>> b
dd.DataFrame<y, divisions=(6, 8, 10)> # doctest: +SKIP
>>> dd.concat([a, b]) # doctest: +SKIP
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.

>>> 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

Different categorieals are unioned

```python
gg.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).dtypes
< categorical[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 or named Series objects with a database-style join.

The join is done on 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] Object to merge with.
right [DataFrame or named Series] Object to merge with.
how [{‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘inner’] Type of merge to be performed.

- 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] Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.
left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [bool, 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 [bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

sort [bool, 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 [tuple of (str, str), default ('_x', '_y')] Suffix to apply to overlapping column names in the left and right side, respectively. To raise an exception on overlapping columns use (False, False).

copy [bool, default True] If False, avoid copy if possible.

indicator [bool or str, 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.

validate [str, optional] 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

- **DataFrame** A DataFrame of the two merged objects.

See also:

- **merge_ordered** Merge with optional filling/interpolation.
- **merge_asof** Merge on nearest keys.
- **DataFrame.join** Similar method using indices.

Notes

Support for specifying index levels as the on, left_on, and right_on parameters was added in version 0.23.0
Support for merging named Series objects was added in version 0.24.0
Examples

```python
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
                    'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
                    'value': [5, 6, 7, 8]})
>>> df1
    lkey  value
0    foo     1
1    bar     2
2    baz     3
3    foo     5
``` 

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```python
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
    lkey  value_x  rkey  value_y
0    foo       1    foo       5
1    foo       1    foo       8
2    foo       5    foo       5
3    foo       5    foo       8
4    bar       2    bar       6
5    baz       3    baz       7
``` 

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```python
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
             suffixes=('_left', '_right'))
    lkey  value_left  rkey  value_right
0    foo          1    foo             5
1    foo          1    foo             8
2    foo          5    foo             5
3    foo          5    foo             8
4    bar          2    bar             6
5    baz          3    baz             7
``` 

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```python
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
  ... 
ValueError: columns overlap but no suffix specified:
  index(["value"], dtype='object')
``` 

dask.dataframe.reshape.get_dummies(data, prefix=None, prefix_sep='__', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=<class 'numpy.uint8'>) 

Convert categorical variable into dummy/indicator variables.

Data must have category dtype to infer result’s columns.

Parameters
**data**  [Series, or DataFrame] For Series, the dtype must be categorical. For DataFrame, at least one column must be categorical.

**prefix**  [string, list of strings, or dict of strings, default None] String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, prefix can be a dictionary mapping column names to prefixes.

**prefix_sep**  [string, default ‘_’] If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with prefix.

**dummy_na**  [bool, default False] Add a column to indicate NaNs, if False NaNs are ignored.

**columns**  [list-like, default None] Column names in the DataFrame to be encoded. If columns is None then all the columns with category dtype will be converted.

**sparse**  [bool, default False] Whether the dummy columns should be sparse or not. Returns SparseDataFrame if data is a Series or if all columns are included. Otherwise returns a DataFrame with some SparseBlocks.

New in version 0.18.2.

**drop_first**  [bool, default False] Whether to get k-1 dummies out of k categorical levels by removing the first level.

**dtype**  [dtype, default np.uint8] Data type for new columns. Only a single dtype is allowed. Only valid if pandas is 0.23.0 or newer.

New in version 0.18.2.

**Returns**

**dummies**  [DataFrame]

**See also:**

pandas.get_dummies

**Examples**

Dask’s version only works with Categorical data, as this is the only way to know the output shape without computing all the data.

```python
>>> import pandas as pd
>>> import dask.dataframe as dd
>>> s = dd.from_pandas(pd.Series(list('abca')), npartitions=2)
>>> dd.get_dummies(s)
Traceback (most recent call last):
  ... Not ImplementedError: `get_dummies` with non-categorical dtypes is not supported...
```

With categorical data:

```python
>>> s = dd.from_pandas(pd.Series(list('abca'), dtype='category'), npartitions=2)
>>> dd.get_dummies(s)  # doctest: +NORMALIZE_WHITESPACE
Dask DataFrame Structure:
    a   b   c
npartitions=2
  0 uint8 uint8 uint8
  2 ...   ...   ...
  3 ...   ...   ...
```

(continues on next page)
Dask Name: get_dummies, 4 tasks

```python
>>> dd.get_dummies(s).compute()
    a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

dask.dataframe.reshape.pivot_table(df, 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

- **data** [DataFrame]
- **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]

The following functions provide access to convert between Dask DataFrames, file formats, and other Dask or Python collections.

File Formats:

- **read_csv(urlpath[, blocksize, collection, ...])** Read CSV files into a Dask.DataFrame
- **read_parquet(path[, columns, filters, ...])** Read ParquetFile into a Dask DataFrame
- **read_hdf(pattern, key[, start, stop, ...])** Read HDF files into a Dask DataFrame
- **read_orc(path[, columns, storage_options])** Read dataframe from ORC file(s)
- **read_json(url_path[, orient, lines, ...])** Create a dataframe from a set of JSON files
- **read_sql_table(table, uri, index_col[, ...])** Create dataframe from an SQL table.
- **read_table(urlpath[, blocksize, collection, ...])** Read delimited files into a Dask.DataFrame
- **read_fwf(urlpath[, blocksize, collection, ...])** Read fixed-width files into a Dask.DataFrame
- **from_bcolz(x[, chunksize, categorize, ...])** Read BColz CTable into a Dask DataFrame
Table 44 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>from_array(x[, chunksize, columns])</code></td>
<td>Read any slicable array into a Dask Dataframe</td>
</tr>
<tr>
<td><code>to_csv(df, filename[, name_function, ...])</code></td>
<td>Store Dask DataFrame to CSV files</td>
</tr>
<tr>
<td><code>to_parquet(df, path[, engine, compression, ...])</code></td>
<td>Store Dask dataframe to Parquet files</td>
</tr>
<tr>
<td><code>to_hdf(df, path, key[, mode, append, ...])</code></td>
<td>Store Dask DataFrame to Hierarchical Data Format (HDF) files</td>
</tr>
</tbody>
</table>

Dask Collections:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>from_delayed(dfs[, meta, divisions, prefix])</code></td>
<td>Create Dask DataFrame from many Dask Delayed objects</td>
</tr>
<tr>
<td><code>from_dask_array(x[, columns, index])</code></td>
<td>Create a Dask DataFrame from a Dask Array.</td>
</tr>
<tr>
<td><code>dask.bag.core.Bag.to_dataframe([meta, columns])</code></td>
<td>Create Dask DataFrame from a Dask Bag.</td>
</tr>
<tr>
<td><code>DataFrame.to_delayed([optimize_graph])</code></td>
<td>Convert into a list of dask.delayed objects, one per partition.</td>
</tr>
<tr>
<td><code>to_records(df)</code></td>
<td>Create Dask Array from a Dask DataFrame</td>
</tr>
<tr>
<td><code>to_bag(df[, index])</code></td>
<td>Create Dask Bag from a Dask DataFrame</td>
</tr>
</tbody>
</table>

Pandas:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>from_pandas(data[, npartitions, chunksize, ...])</code></td>
<td>Construct a Dask DataFrame from a Pandas DataFrame</td>
</tr>
</tbody>
</table>

Locations

For text, CSV, and Apache Parquet formats, data can come from local disk, the Hadoop File System, S3FS, or other sources, by prepending the filenames with a protocol:

```python
>>> 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:

```python
>>> 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 format.

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 that 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)
```

4.9.3 Best Practices

It is easy to get started with Dask DataFrame, but using it well does require some experience. This page contains suggestions for best practices, and includes solutions to common problems.

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.

Reduce, and then use Pandas

Similar to above, even if you have a large dataset there may be a point in your computation where you’ve reduced things to a more manageable level. You may want to switch to Pandas at this point.

```
df = dd.read_parquet('my-giant-file.parquet')
df = df[df.name == 'Alice']             # Select a subsection
result = df.groupby('id').value.mean()  # Reduce to a smaller size
result = result.compute()               # Convert to Pandas dataframe
result...
```

4.9. DataFrame
Pandas Performance Tips Apply to Dask DataFrame

Usual Pandas performance tips like avoiding apply, using vectorized operations, using categoricals, etc., all apply equally to Dask DataFrame. See Modern Pandas by Tom Augspurger for a good read on this topic.

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(column_name)` method. This operation is expensive though, so use it sparingly (see below):

```python
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
```

For more information, see documentation on dataframe partitions.

Avoid Full-Data Shuffling

Setting an index is an important but expensive operation (see above). 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 if 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 intensive process, particularly on a cluster.

So, definitely set the index but try do so infrequently. After you set the index, you may want to persist your data if you are on a cluster:

```python
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. For more information, see the `set_index` docstring.

```python
df2 = df.set_index(d.timestamp, sorted=True)
```

Persist Intelligently

Note: 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:

```python
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. It is a way of telling the cluster that it should start executing 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:

```python
# 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”, and 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 many. 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 by using the `repartition` method:

```python
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 = df.persist()  # 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 another Dask DataFrame of a single partition
3. Joining Dask DataFrames along their indexes

Also, 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:

```
# fast
dd.merge(a, pandas_df)

# fast
dd.merge(a, b, left_index=True, right_index=True)

# half-fast, half-slow
dd.merge(a, b, left_index=True, right_on='id')

# slow
dd.merge(a, b, left_on='id', right_on='id')
```

For more information see *Joins*.

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, and so it 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 parquet files with different engine 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='pyarrow')
```

These libraries can 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.

4.9.4 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:

```python
>>> 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: []
```

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:

```python
>>> 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:

```python
>>> 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. Note that order is important: the order of the names in meta should match the order of the columns
   • 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).

Partitions

Internally, a Dask DataFrame is split into many partitions, where 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:
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:

```python
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`:

```python
df.divisions
```

In these cases, any operation that requires a cleanly partitioned DataFrame with known divisions will have to perform a sort. This can generally achieved by calling `df.set_index(...)`.

### 4.9.5 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:

```python
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:
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 fits 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 and shuffle over the network.

**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 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:

```python
client = Client('scheduler:8786', set_as_default=True)
```

Alternatively, if you prefer to avoid defaults, you can configure the global shuffling method by using the `dask.config.set(shuffle=...)` command. This can be done globally:

```python
dask.config.set(shuffle='tasks')
df.groupby(...).apply(...)```

or as a context manager:

```python
with dask.config.set(shuffle='tasks'):
    df.groupby(...).apply(...)```

In addition, `set_index` also accepts a `shuffle` keyword argument that can be used to select either on-disk or task-based shuffling.
df.set_index(column, shuffle='disk')
df.set_index(column, shuffle='tasks')

4.9.6 Aggregate

Dask supports Pandas’ aggregate syntax to run multiple reductions on the same groups. Common reductions such as max, sum, and mean are directly supported:

```python
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:

```python
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:

```python
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)
```

For example, let’s compute the group-wise extent (maximum - minimum) for a DataFrame.

```python
>>> df = pd.DataFrame({
...     'a': ['a', 'b', 'a', 'a', 'b'],
...     'b': [0, 1, 0, 2, 5],
... })
```

```python
>>> ddf = dd.from_pandas(df, 2)
```

We define the building blocks to find the maximum and minimum of each chunk, and then the maximum and minimum over all the chunks. We finalize by taking the difference between the Series with the maxima and minima

```python
>>> def chunk(grouped):
...     return grouped.max(), grouped.min()

```  

```python
>>> def agg(chunk_maxes, chunk_mins):
...     return chunk_maxes.max(), chunk_mins.min()

```  

```python
>>> def finalize(maxima, minima):
...     return maxima - minima

```  

Finally, we create and use the aggregation
### 4.9.7 Joins

DataFrame joins are a common and expensive computation that benefit from a variety of optimizations in different situations. Understanding how your data is laid out and what you’re trying to accomplish can have a large impact on performance. This documentation page goes through the various different options and their performance impacts.

#### Large to Large Unsorted Joins

In the worst case scenario you have two large tables with many partitions each and you want to join them both along a column that may not be sorted.

This can be slow. In this case Dask DataFrame will need to move all of your data around so that rows with matching values in the joining columns are in the same partition. This large-scale movement can create communication costs, and can require a large amount of memory. If enough memory can not be found then Dask will have to read and write data to disk, which may cause other performance costs.

These problems are solvable, but will be significantly slower than many other operations. They are best avoided if possible.

#### Large to Small Joins

Many join or merge computations combine a large table with one small one. If the small table is either a single partition Dask DataFrame or even just a normal Pandas DataFrame then the computation can proceed in an embarrassingly parallel way, where each partition of the large DataFrame is joined against the single small table. This incurs almost no overhead relative to Pandas joins.

If your smaller table can easily fit in memory, then you might want to ensure that it is a single partition with the following

```python
code
small = small.repartition(npartitions=1)
result = big.merge(small)
```

#### Sorted Joins

The Pandas merge API supports the `left_index=` and `right_index=` options to perform joins on the index. For Dask DataFrames these keyword options hold special significance if the index has known divisions (see `Partitions`). In this case the DataFrame partitions are aligned along these divisions (which is generally fast) and then an embarrassingly parallel Pandas join happens across partition pairs. This is generally relatively fast.

Sorted or indexed joins are a good solution to the large-large join problem. If you plan to join against a dataset repeatedly then it may be worthwhile to set the index ahead of time, and possibly store the data in a format that maintains that index, like Parquet.
left = left.set_index('id').persist()
left.merge(right_one, left_index=True, ...)
left.merge(right_two, left_index=True, ...)
...

### 4.9.8 Indexing into Dask DataFrames

Dask DataFrame supports some of Pandas’ indexing behavior.

<table>
<thead>
<tr>
<th><strong>DataFrame.iloc</strong></th>
<th>Purely integer-location based indexing for selection by position.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataFrame.loc</strong></td>
<td>Purely label-location based indexer for selection by label.</td>
</tr>
</tbody>
</table>

#### Label-based Indexing

Just like Pandas, Dask DataFrame supports label-based indexing with the `.loc` accessor for selecting rows or columns, and `__getitem__` (square brackets) for selecting just columns.

**Note:** To select rows, the DataFrame’s divisions must be known (see *Internal Design* and *Best Practices* for more information.)

```python
>>> import dask.dataframe as dd
>>> import pandas as pd

>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [3, 4, 5]},
                    index=['a', 'b', 'c'])
>>> ddf = dd.from_pandas(df, npartitions=2)

>>> ddf
Dask DataFrame Structure:
  A   B
a  int64  int64
b  ... ... 
... ... ...
Dask Name: from_pandas, 1 tasks

Selecting columns:

```python
>>> ddf[['B', 'A']]
Dask DataFrame Structure:
   B   A
a  int64  int64
b  ... ... 
... ... ...
Dask Name: getitem, 2 tasks

Selecting a single column reduces to a Dask Series:

```python
>>> ddf['A']
Dask Series Structure:
npartitions=1
(continues on next page)```
Slicing rows and (optionally) columns with `.loc`:

```python
>>> ddf.loc[['b', 'c'], ['A']]
Dask DataFrame Structure:
   A
npartitions=1
b int64
c ...
Dask Name: loc, 2 tasks
```

Dask DataFrame supports Pandas’ partial-string indexing:

```python
>>> ts = dd.demo.make_timeseries()
>>> ts
Dask DataFrame Structure:
   id name x y
npartitions=11
2000-01-31 int64 object float64 float64
2000-02-29 ... ... ... ...
... ... ... ...
2000-11-30 ... ... ... ...
2000-12-31 ... ... ... ...
Dask Name: make-timeseries, 11 tasks

>>> ts.loc['2000-02-12']
Dask DataFrame Structure:
   id name x y
npartitions=1
2000-02-12 00:00:00.000000000 int64 object float64 float64
2000-02-12 23:59:59.999999999 ... ... ... ...
Dask Name: loc, 12 tasks
```

Positional Indexing

Dask DataFrame does not track the length of partitions, making positional indexing with `.iloc` inefficient for selecting rows. `DataFrame.iloc()` only supports indexers where the row indexer is `slice(None)` (which : is a shorthand for.)

```python
>>> ddf.iloc[:, [1, 0]]
Dask DataFrame Structure:
   B A
npartitions=1
a ... int64 int64
c ... ...
Dask Name: iloc, 2 tasks
```

Trying to select specific rows with `iloc` will raise an exception:

```python
>>> ddf.iloc[[0, 2], [1]]
Traceback (most recent call last)
```
4.9.9 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:

```python
>>> 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):

```python
>>> 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:

```python
# 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'.
Moreover, 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`:

```python
>>> dtype = {'col': pd.api.types.CategoricalDtype(['a', 'b', 'c'])}
>>> ddf = dd.read_csv(..., dtype=dtype)
```

### 4.9.10 Subclass DataFrames

There are a few projects that subclass or replicate the functionality of Pandas objects:

- GeoPandas: for Geospatial analytics
- PyGDF: for data analysis on GPUs
- ...

These projects may also want to produce parallel variants of themselves with Dask, and may want to reuse some of the code in Dask DataFrame. This document describes how to do this. It is intended for maintainers of these libraries and not for general users.

**Implement `dask`, `name`, `meta`, and `divisions`**

You will need to implement `.meta`, `.dask`, `.divisions`, and `.name` as defined in the `DataFrame design docs`.

**Extend Dispatched Methods**

If you are going to pass around Pandas-like objects that are not normal Pandas objects, then we ask you to extend a few dispatched methods.

#### `make_meta`

This function returns an empty version of one of your non-Dask objects, given a non-empty non-Dask object:

```python
from dask.dataframe import make_meta

@make_meta.register(MyDataFrame)
def make_meta_dataframe(df):
    return df.head(0)

@make_meta.register(MySeries)
def make_meta_series(s):
    return s.head(0)

@make_meta.register(MyIndex)
def make_meta_index(ind):
    return ind[:0]
```

Additionally, you should create a similar function that returns a non-empty version of your non-Dask DataFrame objects filled with a few rows of representative or random data. This is used to guess types when they are not provided.
It should expect an empty version of your object with columns, dtypes, index name, and it should return a non-empty version:

```python
from dask.dataframe.utils import meta_nonempty

@meta_nonempty.register(MyDataFrame)
def meta_nonempty_dataframe(df):
    ...
    return MyDataFrame(..., columns=df.columns,
                        index=MyIndex(..., name=df.index.name)

@meta_nonempty.register(MySeries)
def meta_nonempty_series(s):
    ...

@meta_nonempty.register(MyIndex)
def meta_nonempty_index(ind):
    ...
```

**get_parallel_type**

Given a non-Dask DataFrame object, return the Dask equivalent:

```python
from dask.dataframe.core import get_parallel_type

@get_parallel_type.register(MyDataFrame)
def get_parallel_type_dataframe(df):
    return MyDaskDataFrame

@get_parallel_type.register(MySeries)
def get_parallel_type_series(s):
    return MyDaskSeries

@get_parallel_type.register(MyIndex)
def get_parallel_type_index(ind):
    return MyDaskIndex
```

**concat**

Concatenate many of your non-Dask DataFrame objects together. It should expect a list of your objects (homogeneously typed):

```python
from dask.dataframe.methods import concat_dispatch

@concat_dispatch.register((MyDataFrame, MySeries, MyIndex))
def concat_pandas(dfs, axis=0, join='outer', uniform=False, filter_warning=True):
    ...
```
Extension Arrays

Rather than subclassing Pandas DataFrames, you may be interested in extending Pandas with Extension Arrays.

All of the first-party extension arrays (those implemented in pandas itself) are supported directly by dask.

Developers implementing third-party extension arrays (outside of pandas) will need to do register their ExtensionDtype with Dask so that it works correctly in dask.dataframe.

For example, we’ll register the test-only DecimalDtype from pandas test suite.

```python
from decimal import Decimal
from dask.dataframe.extensions import make_array_nonempty, make_scalar
from pandas.tests.extension.decimal import DecimalArray, DecimalDtype

@make_array_nonempty.register(DecimalDtype)
def _(dtype):
    return DecimalArray._from_sequence([Decimal('0'), Decimal('NaN')],
                                        dtype=dtype)

@make_scalar.register(Decimal)
def _(x):
    return Decimal('1')
```

Internally, Dask will use this to create a small dummy Series for tracking metadata through operations.

```python
>>> make_array_nonempty(DecimalDtype())
<DecimalArray>
[Decimal('0'), Decimal('NaN')]
Length: 2, dtype: decimal
```

So you (or your users) can now create and store a dask DataFrame or Series with your extension array contained within.

```python
>>> from decimal import Decimal
>>> import dask.dataframe as dd
>>> import pandas as pd

>>> from pandas.tests.extension.decimal import DecimalArray

>>> ser = pd.Series(DecimalArray([Decimal('0.0')] * 10))
>>> dser = dd.from_pandas(ser, 3)

Dask Series Structure:

0 decimal 1 ...
1 ...
2 ...
3 ...
4 ...
dtype: decimal
Dask Name: from_pandas, 3 tasks
```

Notice the decimal dtype.

Accessors

Many extension arrays expose their functionality on Series or DataFrame objects using accessors. Dask provides decorators to register accessors similar to pandas. See the pandas documentation on accessors for more.
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. One Dask DataFrame operation triggers many operations on the constituent Pandas DataFrames.

### 4.9.11 Design

Dask DataFrames coordinate many Pandas DataFrames/Series arranged along the index. A Dask DataFrame is partitioned row-wise, grouping rows by index value for efficiency. These Pandas objects may live on disk or on other machines.

### 4.9.12 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:

```python
>>> 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, one triggers computation by calling the `.compute()` method:

```python
>>> df2.compute()
0  2
1  5
Name: x, dtype: int64
```

### 4.9.13 Common Uses and Anti-Uses

Dask DataFrame is used in situations where Pandas is commonly needed, usually when Pandas fails due to data size or computation speed:

- Manipulating large datasets, even when those datasets don’t fit in memory
- Accelerating long computations by using many cores
- Distributed computing on large datasets with standard Pandas operations like groupby, join, and time series computations

Dask DataFrame may not be the best choice in the following situations:

- 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

4.9.14 Scope

Dask DataFrame covers a well-used portion of the Pandas API. The following class of computations works well:

• **Trivially parallelizable operations (fast):**
  - Element-wise 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])]
  - Date time/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')
  - Element-wise operations with different partitions/divisions: df1.x + df2.y
  - Date time resampling: df.resample(...)
  - Rolling averages: df.rolling(...)
  - Pearson’s correlation: 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')

However, 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. Operations that were slow on Pandas, like iterating through row-by-row, remain slow on Dask DataFrame. See DataFrame API documentation for a more extensive list.

4.9.15 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.

When dealing with text data, you may see speedups by switching to the newer distributed scheduler either on a cluster or single machine.

4.10 Delayed

4.10.1 API

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 task graph. 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.
dask.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. Note that this only affects the name of the object wrapped by this call to delayed, and not the output of delayed function calls - for that use dask_key_name= as described below.
- **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:

```python
>>> 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:

```python
>>> @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).

```python
>>> 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.

```python
>>> @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:

```python
>>> import dask
>>> @delayed
... def mul(a, b):
...     return a * b
>>> with dask.config.set(delayed_pure=True):
...     print(mul(1, 2).key == mul(1, 2).key)
True
>>> with dask.config.set(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:

```python
>>> add(1, 2)  # doctest: +SKIP
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.

```python
>>> add(1, 2, dask_key_name='three')  # doctest: +SKIP
>>> add(2, 1, dask_key_name='three')  # doctest: +SKIP
>>> add(2, 2, dask_key_name='four')  # doctest: +SKIP
```

delayed can also be applied to objects to make operations on them lazy:

```python
>>> 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:

```python
>>> a = delayed([1, 2, 3], name='mylist')
>>> a
Delayed('mylist')
```

Delayed results act as a proxy to the underlying object. Many operators are supported:

```python
>>> (a + [1, 2]).compute()
[1, 2, 3, 1, 2]
>>> a[1].compute()
2
```

Method and attribute access also works:

```python
>>> a.count(2).compute()
1
```

Note that if a method doesn’t exist, no error will be thrown until runtime:

```python
>>> res = a.not_a_real_method()
>>> res.compute()
# doctest: +SKIP
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 behavior is not overrideable through the delayed call, but can be modified using other ways as described below.

To invoke an impure attribute or operator, you’d need to use it in a delayed function with pure=False:

```python
>>> 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.

```python
>>> 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:

```python
>>> a.count(2, dask_key_name="count_2")
Delayed('count_2')
>>> with dask.config.set(delayed_pure=True):
```

(continues on next page)
4.10.2 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 the DataFrame (groupby, join, etc.), and then switch back to `dask.delayed` to save our results back to the custom format:

```python
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, and `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.

4.10.3 Best Practices

It is easy to get started with Dask delayed, but using it well does require some experience. This page contains suggestions for best practices, and includes solutions to common problems.

**Call delayed on the function, not the result**

Dask delayed operates on functions like `dask.delayed(f)(x, y)`, not on their results like `dask.delayed(f(x, y))`. When you do the latter, Python first calculates `f(x, y)` before Dask has a chance to step in.

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td># This executes immediately <code>dask.delayed(f(x, y))</code></td>
<td># This makes a delayed function, acting lazily <code>dask.delayed(f)(x, y)</code></td>
</tr>
</tbody>
</table>
Compute on lots of computation at once

To improve parallelism, you want to include lots of computation in each compute call. Ideally, you want to make many `dask.delayed` calls to define your computation and then call `dask.compute` only at the end. It is ok to call `dask.compute` in the middle of your computation as well, but everything will stop there as Dask computes those results before moving forward with your code.

### Don’t vs. Do

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td># Avoid calling compute repeatedly</td>
<td># Collect many calls for one compute</td>
</tr>
<tr>
<td>results = []</td>
<td>results = []</td>
</tr>
<tr>
<td>for x in L:</td>
<td>for x in L:</td>
</tr>
<tr>
<td>y = dask.delayed(f)(x)</td>
<td>y = dask.delayed(f)(x)</td>
</tr>
<tr>
<td>results.append(y.compute())</td>
<td>results.append(y)</td>
</tr>
<tr>
<td>results</td>
<td>results = dask.compute(*results)</td>
</tr>
</tbody>
</table>

Calling `y.compute()` within the loop would await the result of the computation every time, and so inhibit parallelism.

### Don’t mutate inputs

Your functions should not change the inputs directly.

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td># Mutate inputs in functions</td>
<td># Return new values or copies</td>
</tr>
<tr>
<td>@dask.delayed</td>
<td>@dask.delayed</td>
</tr>
<tr>
<td>def f(x):</td>
<td>def f(x):</td>
</tr>
<tr>
<td>x += 1</td>
<td>x = x + 1</td>
</tr>
<tr>
<td>return x</td>
<td>return x</td>
</tr>
</tbody>
</table>

If you need to use a mutable operation, then make a copy within your function first:

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>@dask.delayed</td>
<td>@dask.delayed</td>
</tr>
<tr>
<td>def f(x):</td>
<td>def f(x):</td>
</tr>
<tr>
<td>x = copy(x)</td>
<td>x = x + 1</td>
</tr>
<tr>
<td>x += 1</td>
<td>return x</td>
</tr>
</tbody>
</table>

### Avoid global state

Ideally, your operations shouldn’t rely on global state. Using global state might work if you only use threads, but when you move to multiprocessing or distributed computing then you will likely encounter confusing errors.
Don’t

L = []

# This references global variable L

@dask.delayed
def f(x):
    L.append(x)

Don’t rely on side effects

Delayed functions only do something if they are computed. You will always need to pass the output to something that eventually calls compute.

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td># Forget to call compute</td>
<td># Ensure delayed tasks are computed</td>
</tr>
<tr>
<td>dask.delayed(f)(1, 2, 3)</td>
<td>x = dask.delayed(f)(1, 2, 3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>dask.compute(x, ...)</td>
<td></td>
</tr>
</tbody>
</table>

In the first case here, nothing happens, because `compute()` is never called.

Break up computations into many pieces

Every `dask.delayed` function call is a single operation from Dask’s perspective. You achieve parallelism by having many delayed calls, not by using only a single one: Dask will not look inside a function decorated with `@dask.delayed` and parallelize that code internally. To accomplish that, it needs your help to find good places to break up a computation.
The first version only has one delayed task, and so cannot parallelize.

## Avoid too many tasks

Every delayed task has an overhead of a few hundred microseconds. Usually this is ok, but it can become a problem if you apply dask.delayed too finely. In this case, it’s often best to break up your many tasks into batches or use one of the Dask collections to help you.

```python
Don’t

# One giant task

def load(filename):
    ...

def process(filename):
    ...

def save(filename):
    ...

@dask.delayed
def f(filenames):
    results = []
    for filename in filenames:
        data = load(filename)
        data = process(data)
        result = save(data)

    return results

dask.compute(f(filenames))

# Too many tasks

results = []
for x in range(1000000000):
    y = dask.delayed(f)(x)
    results.append(y)

# Use collections

import dask.bag as db
b = db.from_sequence(1000000000, npartitions=1000)
b = b.map(f)
...
```

Here we use dask.bag to automatically batch applying our function. We could also have constructed our own batching as follows

```python
Don’t

# Too many tasks

results = []
for x in range(1000000000):
    y = dask.delayed(f)(x)
    results.append(y)

# Use collections

import dask.bag as db
b = db.from_sequence(1000000000, npartitions=1000)
b = b.map(f)
...
```

(continues on next page)
sub_results.append(f(x))
return sub_results

batches = []
for i in range(0, 1000000000, 1000000):
    result_batch = dask.delayed(batch, range(i, i + 1000000))
    batches.append(result_batch)

Here we construct batches where each delayed function call computes for many data points from the original input.

**Avoid calling delayed within delayed functions**

Often, if you are new to using Dask delayed, you place `dask.delayed` calls everywhere and hope for the best. While this may actually work, it’s usually slow and results in hard-to-understand solutions. Usually you never call `dask.delayed` within `dask.delayed` functions.

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td># Delayed function calls delayed</td>
<td># Normal function calls delayed</td>
</tr>
<tr>
<td>@dask.delayed</td>
<td></td>
</tr>
<tr>
<td>def process_all(L):</td>
<td>def process_all(L):</td>
</tr>
<tr>
<td>result = []</td>
<td>result = []</td>
</tr>
<tr>
<td>for x in L:</td>
<td>for x in L:</td>
</tr>
<tr>
<td>y = dask.delayed(f)(x)</td>
<td>y = dask.delayed(f)(x)</td>
</tr>
<tr>
<td>result.append(y)</td>
<td>result.append(y)</td>
</tr>
<tr>
<td>return result</td>
<td>return result</td>
</tr>
</tbody>
</table>

Because the normal function only does delayed work it is very fast and so there is no reason to delay it.

**Don’t call dask.delayed on other Dask collections**

When you place a Dask array or Dask DataFrame into a delayed call, that function will receive the NumPy or Pandas equivalent. Beware that if your array is large, then this might crash your workers.

Instead, it’s more common to use methods like `da.map_blocks`

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td># Call delayed functions on Dask_</td>
<td># Use mapping methods if applicable</td>
</tr>
<tr>
<td>→collections</td>
<td></td>
</tr>
<tr>
<td>import dask.dataframe as dd</td>
<td>import dask.dataframe as dd</td>
</tr>
<tr>
<td>df = dd.read_csv('/path/to/*.csv')</td>
<td>df = dd.read_csv('/path/to/*.csv')</td>
</tr>
<tr>
<td>dask.delayed(train)(df)</td>
<td>df.map_partitions(train)</td>
</tr>
</tbody>
</table>

Alternatively, if the procedure doesn’t fit into a mapping, you can always turn your arrays or dataframes into many delayed objects, for example
partitions = df.to_delayed()
delayed_values = [dask.delayed(train)(part)
    for part in partitions]

However, if you don’t mind turning your Dask array/DataFrame into a single chunk, then this is ok.
dask.delayed(train)(..., y=df.sum())

Avoid repeatedly putting large inputs into delayed calls

Every time you pass a concrete result (anything that isn’t delayed) Dask will hash it by default to give it a name. This
is fairly fast (around 500 MB/s) but can be slow if you do it over and over again. Instead, it is better to delay your data
as well.

This is especially important when using a distributed cluster to avoid sending your data separately for each function
call.

<table>
<thead>
<tr>
<th>Don’t</th>
<th>Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = np.array(...) # some large array</td>
<td>x = np.array(...) # some large array</td>
</tr>
<tr>
<td>results = [dask.delayed(train)(x, i)</td>
<td>x = dask.delayed(x) # delay the data, hashing once</td>
</tr>
<tr>
<td>for i in range(1000)]</td>
<td>for i in range(1000)]</td>
</tr>
</tbody>
</table>

Every call to `dask.delayed(train)(x, ...)` has to hash the NumPy array `x`, which slows things down.

Do

x = np.array(...) # some large array
x = dask.delayed(x) # delay the data, hashing once
results = [dask.delayed(train)(x, i) for i in range(1000)]

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:

```python
>>> x = dask.delayed(inc)(1)
>>> y = dask.delayed(inc)(2)
>>> z = dask.delayed(add)(x, y)
>>> z.compute()
5
>>> z.visualize()
```

4.10.4 Example

Sometimes we face problems that are parallelizable, but don’t fit into high-level abstractions like Dask Array or Dask
DataFrame. Consider the following example:
There is clearly parallelism in this problem (many of the inc, double, and add functions can evaluate independently), but it’s not clear how to convert this to a big array or big DataFrame computation.

As written, this code runs sequentially in a single thread. However, we see that a lot of this could be executed in parallel.

The Dask delayed function decorates your functions so that they operate lazily. Rather than executing your function immediately, it will defer execution, placing the function and its arguments into a task graph.

We slightly modify our code by wrapping functions in delayed. This delays the execution of the function and generates a Dask graph instead:

```python
import dask

data = [1, 2, 3, 4, 5]
output = []
for x in data:
a = dask.delayed(inc)(x)
b = dask.delayed(double)(x)
c = dask.delayed(add)(a, b)
output.append(c)
total = dask.delayed(sum)(output)
```

We can now compute this lazy result to execute the graph in parallel:

```python
total.visualize()  # see image to the right
```

We used the dask.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.)
4.10.5 Decorator

It is also common to see the delayed function used as a decorator. Here is a reproduction of our original problem as a parallel code:

```python
import dask
@dask.delayed
def inc(x):
    return x + 1
@dask.delayed
def double(x):
    return x + 2
@dask.delayed
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)
```

```
45
```

4.10.6 Real time

Sometimes you want to create and destroy work during execution, launch tasks from other tasks, etc. For this, see the Futures interface.

4.10.7 Best Practices

For a list of common problems and recommendations see Delayed Best Practices.

4.11 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).
4.11.1 Start Dask Client

You must start a `Client` to use the futures interface. This tracks state among the various worker processes or threads:

```python
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.

4.11.2 Submit Tasks

- `Client.submit(func, *args, **kwargs)`: Submit a function application to the scheduler
- `Client.map(func, *iterables, **kwargs)`: Map a function on a sequence of arguments
- `Future.result([timeout])`: Wait until computation completes, gather result to local process.

You can submit individual tasks using the `submit` method:

```python
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
```

The `submit` function returns a `Future`, which refers to a remote result. This result may not yet be completed:

```python
>>> 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:

```python
>>> 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:

```python
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:

```python
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.

### 4.11.3 Move Data

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Future.result([timeout])</code></td>
<td>Wait until computation completes, gather result to local process.</td>
</tr>
<tr>
<td><code>Client.gather(futures[, errors, maxsize,...])</code></td>
<td>Gather futures from distributed memory</td>
</tr>
<tr>
<td><code>Client.scatter(data[, workers, broadcast,...])</code></td>
<td>Scatter data into distributed memory</td>
</tr>
</tbody>
</table>

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:

```python
>>> c.result()
12
```

You can gather many results concurrently using the `Client.gather` method. This can be more efficient than calling `.result()` on each future sequentially:

```python
>>> # 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:

```python
>>> 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:

```python
>>> remote_df = client.scatter(df)
>>> remote_df
<Future: status: finished, type: DataFrame, key: bbd0ca93589c56ea14af49c49c470006e>

>>> 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:

```python
>>> client.scatter([1, 2, 3])
<Future: status: finished, type: int, key: c0a8a20f903a4915b94db8de3e3a63195>,
<Future: status: finished, type: int, key: 58e78e1b34eb49a68c65b54815d1b158>,
<Future: status: finished, type: int, key: d3395e15f605bc35ab1bac6341a285e2>]
```

### 4.11.4 References, Cancellation, and Exceptions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Future.cancel(**kwargs)</code></td>
<td>Cancel request to run this future</td>
</tr>
<tr>
<td><code>Future.exception([timeout])</code></td>
<td>Return the exception of a failed task</td>
</tr>
</tbody>
</table>

Continued on next page
Future.traceback([timeout]) Return the traceback of a failed task
Client.cancel(futures[, asynchronous, force]) 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:

```python
>>> 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:

```python
>>> 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:

```python
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()
ZeroDivisionError: division by zero
```

All futures that depend on an erred future also err with the same exception:

```python
>>> 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.

### 4.11.5 Waiting on Futures

| as_completed([futures, loop, with_results,...]) | Return futures in the order in which they complete |
| wait([fs[, timeout, return_when]]) | Wait until all/any futures are finished |

You can wait on a future or collection of futures using the `wait` function:

```python
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:

```python
from dask.distributed import as_completed

(continues on next page)```
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, result in as_completed(futures, with_results=True):
    # y = future.result()  # don't need this
...

Or collect all futures in batches that had arrived since the last iteration:

for batch in as_completed(futures, with_results=True).batches():
    for future, result in batch:
        ...

Additionally, for iterative algorithms, you can add more futures into the as_completed iterator during iteration:

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

4.11.6 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:

```python
>>> a = client.submit(load, filename)
>>> b = client.submit(process, a)
>>> c = client.submit(write, b, 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:

```python
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:

```python
def process(filename):
    out_filename = 'out-' + filename
```
a = client.submit(load, filename)
b = client.submit(process, a)
c = client.submit(write, b, 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)

4.11.7 Submit Tasks from Tasks

| get_client([address, timeout, resolve_address]) | Get a client while within a task. |
| rejoin() | Have this thread rejoin the ThreadPoolExecutor |
| secede() | Have this task secede from the worker’s thread pool |

This is an advanced feature and is rarely necessary in the common case.

Tasks can launch other tasks by getting their own client. This enables complex and highly dynamic workloads:

```python
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:

```python
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:

```python
from dask.distributed import get_client, secede
```
def monitor(device):
    client = get_client()
    secede()  # remove this task from the thread pool
    while True:
        data = device.read_data()
        future = client.submit(process, data)
        fire_and_forget(future)

If you intend to do more work in the same thread after waiting on client work, you may want to explicitly block until
the thread is able to rejoin the thread pool. This allows some control over the number of threads that are created and
stops too many threads from being active at once, over-saturating your hardware:

def f(n):
    # assume that this runs as a task
    client = get_client()
    secede()  # secede while we wait for results to come back
    futures = client.map(func, range(n))
    results = client.gather(futures)
    rejoin()  # block until a slot is open in the thread pool
    result = analyze(results)
    return result

Alternatively, you can just use the normal compute function within a task. This will automatically call secede and
rejoin appropriately:

def f(name, fn):
    df = dd.read_csv(fn)  # note that this is a dask collection
    result = df[df.name == name].count()

    # This calls secede
    # Then runs the computation on the cluster (including this worker)
    # Then blocks on rejoin, and finally delivers the answer
    result = result.compute()

    return result

4.11.8 Coordination Primitives

<table>
<thead>
<tr>
<th>Queue(name, client, maxsize)</th>
<th>Distributed Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable(name, client, maxsize)</td>
<td>Distributed Global Variable</td>
</tr>
<tr>
<td>Lock(name, client)</td>
<td>Distributed Centralized Lock</td>
</tr>
<tr>
<td>PUB(name, worker, client)</td>
<td>Publish data with Publish-Subscribe pattern</td>
</tr>
<tr>
<td>Sub(name, worker, client)</td>
<td>Subscribe to a Publish/Subscribe topic</td>
</tr>
</tbody>
</table>

Sometimes situations arise where tasks, workers, or clients need to coordinate with each other in ways beyond normal
task scheduling with futures. In these cases Dask provides additional primitives to help in complex situations.

Dask provides distributed versions of coordination primitives like locks, queues, global variables, and pub-sub systems
that, where appropriate, match their in-memory counterparts. These can be used to control access to external resources,
track progress of ongoing computations, or share data in side-channels between many workers, clients, and tasks
sensibly.

These features are rarely necessary for common use of Dask. We recommend that beginning users stick with using
the simpler futures found above (like `Client.submit` and `Client.gather`) rather than embracing needlessly complex techniques.

**Queues**

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:

```python
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:

```python
def func(x):
    try:
        ...
    except Exception as e:
        error_queue.put(str(e))

error_queue = Queue()
```

Queues are mediated by the central scheduler, and so they are not ideal for sending large amounts of data (everything you send will be routed through a central point). They are well suited to move around small bits of metadata, or futures. These futures may point to much larger pieces of data safely:

```python
>>> x = ... # my large numpy array
# Don't do this!
>>> q.put(x)

# Do this instead
>>> future = client.scatter(x)
>>> q.put(future)

# Or use futures for metadata
>>> q.put({'status': 'OK', 'stage': 1234})
```
If you’re looking to move large amounts of data between workers, then you might also want to consider the Pub/Sub system described a few sections below.

**Global Variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Distributed Global Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>([name, client, maxsize])</td>
<td>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:</td>
</tr>
</tbody>
</table>

```python
>>> var = Variable('stopping-criterion')
>>> var.set(False)
>>> var.get()
False
```

This is often used to signal stopping criteria or current parameters between clients.

If you want to share large pieces of information, then scatter the data first:

```python
>>> parameters = np.array(...)  
>>> future = client.scatter(parameters)  
>>> var.set(future)
```

**Locks**

<table>
<thead>
<tr>
<th>Lock</th>
<th>Distributed Centralized Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>([name, client])</td>
<td>You can also hold onto cluster-wide locks using the Lock object. Dask Locks have the same API as normal <code>threading.Lock</code> objects, except that they work across the cluster:</td>
</tr>
</tbody>
</table>

```python
from dask.distributed import Lock
lock = Lock()

with lock:
    # access protected resource
```

You can manage several locks at the same time. Lock can either be given a consistent name or you can pass the lock object around itself.

Using a consistent name is convenient when you want to lock some known named resource:

```python
from dask.distributed import Lock

def load(fn):
    with Lock('the-production-database'):
        # read data from filename using some sensitive source
        return ...

futures = client.map(load, filenames)
```

Passing around a lock works as well and is easier when you want to create short-term locks for a particular situation:
from dask.distributed import Lock
lock = Lock()

def load(fn, lock=None):
    with lock:
        # read data from filename using some sensitive source
        return ...

futures = client.map(load, filenames, lock=lock)

This can be useful if you want to control concurrent access to some external resource like a database or un-thread-safe library.

**Publish-Subscribe**

<table>
<thead>
<tr>
<th>Pub(name[, worker, client])</th>
<th>Publish data with Publish-Subscribe pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub(name[, worker, client])</td>
<td>Subscribe to a Publish/Subscribe topic</td>
</tr>
</tbody>
</table>

Dask implements the Publish Subscribe pattern, providing an additional channel of communication between ongoing tasks.

class distributed.Pub(name, worker=None, client=None)

Publish data with Publish-Subscribe pattern

This allows clients and workers to directly communicate data between each other with a typical Publish-Subscribe pattern. This involves two components,

Pub objects, into which we put data:

```python
>>> pub = Pub('my-topic')
>>> pub.put(123)
```

And Sub objects, from which we collect data:

```python
>>> sub = Sub('my-topic')
>>> sub.get()
123
```

Many Pub and Sub objects can exist for the same topic. All data sent from any Pub will be sent to all Sub objects on that topic that are currently connected. Pub’s and Sub’s find each other using the scheduler, but they communicate directly with each other without coordination from the scheduler.

Pubs and Subs use the central scheduler to find each other, but not to mediate the communication. This means that there is very little additional latency or overhead, and they are appropriate for very frequent data transfers. For context, most data transfer first checks with the scheduler to find which workers should participate, and then does direct worker-to-worker transfers. This checking in with the scheduler provides some stability guarantees, but also adds in a few extra network hops. PubSub doesn’t do this, and so is faster, but also can easily drop messages if Pubs or Subs disappear without notice.

When using a Pub or Sub from a Client all communications will be routed through the scheduler. This can cause some performance degradation. PubS an Subs only operate at top-speed when they are both on workers.

**Parameters**

- name: object (msgpack serializable)  The name of the group of Pubs and Subs on which to participate
See also:

Sub

Examples

```python
>>> pub = Pub('my-topic')
>>> sub = Sub('my-topic')
>>> pub.put([1, 2, 3])
>>> sub.get()
[1, 2, 3]
```

You can also use sub within a for loop:

```python
>>> for msg in sub: # doctest: +SKIP
...    print(msg)
```

or an async for loop

```python
>>> async for msg in sub: # doctest: +SKIP
...    print(msg)
```

Similarly the .get method will return an awaitable if used by an async client or within the IOLoop thread of a worker

```python
>>> await sub.get() # doctest: +SKIP
```

You can see the set of connected worker subscribers by looking at the .subscribers attribute:

```python
>>> pub.subscribers
{'tcp://...': {}, 'tcp://...': {}}
```

**put** *(msg)*

Publish a message to all subscribers of this topic

## 4.11.9 Actors

**Note:** This is an advanced feature and is rarely necessary in the common case.

**Note:** This is an experimental feature and is subject to change without notice.

Actors allow workers to manage rapidly changing state without coordinating with the central scheduler. This has the advantage of reducing latency (worker-to-worker roundtrip latency is around 1ms), reducing pressure on the centralized scheduler (workers can coordinate actors entirely among each other), and also enabling workflows that require stateful or in-place memory manipulation.

However, these benefits come at a cost. The scheduler is unaware of actors and so they don’t benefit from diagnostics, load balancing, or resilience. Once an actor is running on a worker it is forever tied to that worker. If that worker becomes overburdened or dies, then there is no opportunity to recover the workload.

*Because Actors avoid the central scheduler they can be high-performing, but not resilient.*

## 4.11. Futures
Example: Counter

An actor is a class containing both state and methods that is submitted to a worker:

```python
class Counter:
    n = 0

    def __init__(self):
        self.n = 0

    def increment(self):
        self.n += 1
        return self.n

from dask.distributed import Client
client = Client()
future = client.submit(Counter, actor=True)
counter = future.result()
```

Method calls on this object produce ActorFutures, which are similar to normal Futures, but interact only with the worker holding the Actor:

```python
>>> future = counter.increment()
>>> future
<ActorFuture>
```

Attribute access is synchronous and blocking:

```python
>>> counter.n
1
```

Example: Parameter Server

```python
import numpy as np

from dask.distributed import Client
client = Client(processes=False)

class ParameterServer:
    def __init__(self):
        self.data = dict()

    def put(self, key, value):
        self.data[key] = value

    def get(self, key):
        return self.data[key]

ps_future = client.submit(ParameterServer, actor=True)
```

(continues on next page)
ps = ps_future.result()
ps.put('parameters', np.random.random(1000))

```python
def train(batch, ps):
    params = ps.get('parameters')
    for batch in batches:
        # Asynchronous Operation
All operations that require talking to the remote worker are awaitable:

```async def f():
    future = client.submit(Counter, actor=True)
    counter = await future  # gather actor object locally
    counter.increment()  # send off a request asynchronously
    await counter.increment()  # or wait until it was received
    n = await counter.n  # attribute access also must be awaited

```

### 4.11.10 API

#### Client

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Client([address, loop, timeout, ...])</code></td>
<td>Connect to and drive computation on a distributed Dask cluster</td>
</tr>
<tr>
<td><code>Client.cancel(futures[, asynchronous, force])</code></td>
<td>Cancel running futures</td>
</tr>
<tr>
<td><code>Client.compute(collections[, sync, ...])</code></td>
<td>Compute dask collections on cluster</td>
</tr>
<tr>
<td><code>Client.gather(futures[, errors, maxsize, ...])</code></td>
<td>Gather futures from distributed memory</td>
</tr>
<tr>
<td><code>Client.get(dsk, keys[, restrictions, ...])</code></td>
<td>Compute dask graph</td>
</tr>
<tr>
<td><code>Client.get_dataset(name, **kwargs)</code></td>
<td>Get named dataset from the scheduler</td>
</tr>
<tr>
<td><code>Client.get_executor(**kwargs)</code></td>
<td>Return a concurrent.futures Executor for submitting tasks on this Client</td>
</tr>
<tr>
<td><code>Client.has_what([workers])</code></td>
<td>Which keys are held by which workers</td>
</tr>
<tr>
<td><code>Client.list_datasets(**kwargs)</code></td>
<td>List named datasets available on the scheduler</td>
</tr>
<tr>
<td><code>Client.map(func, *iterables, **kwargs)</code></td>
<td>Map a function on a sequence of arguments</td>
</tr>
<tr>
<td><code>Client.ncores([workers])</code></td>
<td>The number of threads/cores available on each worker node</td>
</tr>
<tr>
<td><code>Client.persist(collections[, ...])</code></td>
<td>Persist dask collections on cluster</td>
</tr>
<tr>
<td><code>Client.publish_dataset(*args, **kwargs)</code></td>
<td>Publish named datasets to scheduler</td>
</tr>
<tr>
<td><code>Client.rebalance([futures, workers])</code></td>
<td>Rebalance data within network</td>
</tr>
<tr>
<td><code>Client.replicate(futures[, n, workers, ...])</code></td>
<td>Set replication of futures within network</td>
</tr>
<tr>
<td><code>Client.restart(**kwargs)</code></td>
<td>Restart the distributed network</td>
</tr>
<tr>
<td><code>Client.run(function, *args, **kwargs)</code></td>
<td>Run a function on all workers outside of task scheduling system</td>
</tr>
<tr>
<td><code>Client.run_on_scheduler(function, *args, ...)</code></td>
<td>Run a function on the scheduler process</td>
</tr>
<tr>
<td><code>Client.scatter(data[, workers, broadcast, ...])</code></td>
<td>Scatter data into distributed memory</td>
</tr>
<tr>
<td><code>Client.shutdown(*args, **kwargs)</code></td>
<td>Deprecated, see close instead</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 61 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Client.scheduler_info(**kwargs)</code></td>
<td>Basic information about the workers in the cluster</td>
</tr>
<tr>
<td><code>Client.shutdown(*args, **kwargs)</code></td>
<td>Deprecated, see close instead</td>
</tr>
<tr>
<td><code>Client.start_ipython_workers([workers, ...])</code></td>
<td>Start IPython kernels on workers</td>
</tr>
<tr>
<td><code>Client.start_ipython_scheduler([magic_name, ...])</code></td>
<td>Start IPython kernel on the scheduler</td>
</tr>
<tr>
<td><code>Client.submit(func, *args, **kwargs)</code></td>
<td>Submit a function application to the scheduler</td>
</tr>
<tr>
<td><code>Client.unpublish_dataset(name, **kwargs)</code></td>
<td>Remove named datasets from scheduler</td>
</tr>
<tr>
<td><code>Client.upload_file(filename, **kwargs)</code></td>
<td>Upload local package to workers</td>
</tr>
<tr>
<td><code>Client.who_has([futures])</code></td>
<td>The workers storing each future’s data</td>
</tr>
</tbody>
</table>

### Future

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Future(key[, client, inform, state])</code></td>
<td>A remotely running computation</td>
</tr>
<tr>
<td><code>Future.add_done_callback(fn)</code></td>
<td>Call callback on future when callback has finished</td>
</tr>
<tr>
<td><code>Future.cancel(**kwargs)</code></td>
<td>Cancel request to run this future</td>
</tr>
<tr>
<td><code>Future.cancelled()</code></td>
<td>Returns True if the future has been cancelled</td>
</tr>
<tr>
<td><code>Future.done()</code></td>
<td>Is the computation complete?</td>
</tr>
<tr>
<td><code>Future.exception([timeout])</code></td>
<td>Return the exception of a failed task</td>
</tr>
<tr>
<td><code>Future.result([timeout])</code></td>
<td>Wait until computation completes, gather result to local process.</td>
</tr>
<tr>
<td><code>Future.traceback([timeout])</code></td>
<td>Return the traceback of a failed task</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as_completed([futures, loop, with_results, ...])</code></td>
<td>Return futures in the order in which they complete</td>
</tr>
<tr>
<td><code>fire_and_forget(obj)</code></td>
<td>Run tasks at least once, even if we release the futures</td>
</tr>
<tr>
<td><code>get_client([address, timeout, resolve_address])</code></td>
<td>Get a client while within a task.</td>
</tr>
<tr>
<td><code>secede()</code></td>
<td>Have this task secede from the worker’s thread pool</td>
</tr>
<tr>
<td><code>rejoin()</code></td>
<td>Have this thread rejoin the ThreadPoolExecutor</td>
</tr>
<tr>
<td><code>wait(fs[, timeout, return_when])</code></td>
<td>Wait until all/any futures are finished</td>
</tr>
</tbody>
</table>

```python
distributed.as_completed (futures=None, loop=None, with_results=False, raise_errors=True)
```

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

#### Parameters

- **futures**: `Collection of futures` A list of Future objects to be iterated over in the order in which they complete
- **with_results**: `bool` (False) Whether to wait and include results of futures as well; in this case `as_completed` yields a tuple of (future, result)
- **raise_errors**: `bool` (True) Whether we should raise when the result of a future raises an exception; only affects behavior when `with_results=True`. 

---

760 Chapter 4. Index
Examples

```python
>>> x, y, z = client.map(inc, [1, 2, 3])  # doctest: +SKIP
>>> for future in as_completed([x, y, z]):  # doctest: +SKIP
...     print(future.result())  # doctest: +SKIP
3
2
4
```

Add more futures during computation

```python
>>> x, y, z = client.map(inc, [1, 2, 3])  # doctest: +SKIP
>>> ac = as_completed([x, y, z])  # doctest: +SKIP
>>> for future in ac:
...     print(future.result())  # doctest: +SKIP
...     if random.random() < 0.5:
...         ac.add(c.submit(double, future))  # doctest: +SKIP
4
2
8
3
6
12
24
```

Optionally wait until the result has been gathered as well

```python
>>> ac = as_completed([x, y, z], with_results=True)  # doctest: +SKIP
>>> for future, result in ac:
...     print(result)  # doctest: +SKIP
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**

```python
>>> fire_and_forget(client.submit(func, *args))  # doctest: +SKIP
```

distributed.**get_client**(address=None, timeout=3, resolve_address=True)

Get a client while within a task.

This client connects to the same scheduler to which the worker is connected
Parameters

- **address** [str, optional] The address of the scheduler to connect to. Defaults to the scheduler the worker is connected to.
- **timeout** [int, default 3] Timeout (in seconds) for getting the Client
- **resolve_address** [bool, default True] Whether to resolve address to its canonical form.

Returns

Client

See also:

get_worker, worker_client, secede

Examples

```python
>>> 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)  # doctest: +SKIP
>>> future.result()  # doctest: +SKIP
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

```python
>>> 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.rejoin()

Have this thread rejoin the ThreadPoolExecutor

This will block until a new slot opens up in the executor. The next thread to finish a task will leave the pool to allow this one to join.

See also:

secede leave the thread pool
distributed.wait (fs, timeout=None, return_when='ALL_COMPLETED')
Wait until all/any futures are finished

Parameters

- **fs**: list of futures
- **timeout**: number, optional  Time in seconds after which to raise a dask.distributed. TimeoutError

---

Named tuple of completed, not completed

class distributed.Client (address=None, loop=None, timeout='__no_default__', set_as_default=True, scheduler_file=None, security=None, asynchronous=False, name=None, heartbeat_interval=None, serializers=None, deserializers=None, extensions=[<class 'distributed.pubsub.PubSubClientExtension'>], direct_to_workers=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 using this client within async/await functions or within Tornado gen.coroutines. Otherwise this should remain False for normal use.
- **name**: string (optional)  Gives the client a name that will be included in logs generated on the scheduler for matters relating to this client
- **direct_to_workers**: bool (optional)  Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary.
- **heartbeat_interval**: int  Time in milliseconds between heartbeats to scheduler
- ****kwargs**: If you do not pass a scheduler address, Client will create a LocalCluster object, passing any extra keyword arguments.

See also:

- distributed.scheduler.Scheduler  Internal scheduler
- distributed.deploy.local.LocalCluster
Examples

Provide cluster’s scheduler node address on initialization:

```python
>>> client = Client('127.0.0.1:8786')  # doctest: +SKIP
```

Use submit method to send individual computations to the cluster

```python
>>> a = client.submit(add, 1, 2)  # doctest: +SKIP
>>> b = client.submit(add, 10, 20)  # doctest: +SKIP
```

Continue using submit or map on results to build up larger computations

```python
>>> c = client.submit(add, a, b)  # doctest: +SKIP
```

Gather results with the gather method.

```python
>>> client.gather(c)  # doctest: +SKIP
33
```

You can also call Client with no arguments in order to create your own local cluster.

```python
>>> client = Client()  # makes your own local "cluster"  # doctest: +SKIP
```

Extra keywords will be passed directly to LocalCluster

```python
>>> client = Client(processes=False, threads_per_worker=1)  # doctest: +SKIP
```

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:

```python
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

```python
>>> df = dd.read_parquet(...).persist()  # doctest: +SKIP
>>> client.call_stack(df)  # call on collections
```
cancel \( (\text{futures}, \text{asynchronous}=\text{None}, \text{force}=\text{False}) \)

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 \( (\text{timeout}=\text{'\_\_no\_default\_\_'})) \)

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 \( (\text{collections}, \text{sync}=\text{False}, \text{optimize_graph}=\text{True}, \text{workers}=\text{None}, \text{allow_other_workers}=\text{False}, \text{resources}=\text{None}, \text{retries}=0, \text{priority}=0, \text{fifo_timeout}=\text{'60s'}, \text{actors}=\text{None}, **\text{kwargs}) \)

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
- **priority**: Number Optional prioritization of task. Zero is default. Higher priorities take precedence
- **fifo_timeout**: timedelta str (defaults to '60s') Allowed amount of time between calls to consider the same priority
- ****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

---

4.11. Futures 765
Examples

```python
>>> from dask import delayed
>>> from operator import add

>>> x = delayed(add)(1, 2)
>>> y = delayed(add)(x, x)

```

Also support single arguments

```python
>>> xx = client.compute(x)  # doctest: +SKIP
```

### 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

- **direct**: boolean
  Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary. This can also be set when creating the Client.

- **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

```python
>>> from operator import add  # doctest: +SKIP
>>> c = Client('127.0.0.1:8787')  # doctest: +SKIP
>>> x = c.submit(add, 1, 2)  # doctest: +SKIP
>>> c.gather(x)  # doctest: +SKIP
3
>>> c.gather([x, [x], x])  # support lists and dicts  # doctest: +SKIP
[3, [3], 3]
```
```python
>>> seq = c.gather(iter([x, x]))  # support iterators  # doctest: +SKIP
>>> next(seq)  # doctest: +SKIP
3
```

**get** *(dsk, keys, restrictions=None, loose_restrictions=None, resources=None, sync=True, asynchronous=None, direct=None, retries=None, priority=0, fifo_timeout='60s', actors=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
  - retries: int (default to 0) Number of allowed automatic retries if computing a result fails
  - priority: Number Optional prioritization of task. Zero is default. Higher priorities take precedence
  - sync: bool (optional) Returns Futures if False or concrete values if True (default).
  - direct: bool Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary. This can also be set when creating the Client.

**See also:**

- **Client.compute** Compute asynchronous collections

**Examples**

```python
>>> from operator import add  # doctest: +SKIP
>>> c = Client('127.0.0.1:8787')  # doctest: +SKIP
>>> c.get({'x': (add, 1, 2)}, 'x')  # doctest: +SKIP
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

4.11. Futures 767
Parameters

keys: key or list  Key to access. If a list then gets within a nested collection

default: optional  If the key does not exist then return this value instead. If not provided
then this raises a KeyError if the key is not present

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_task_stream (start=None, stop=None, count=None, plot=False, filename='task-stream.html')
Get task stream data from scheduler

This collects the data present in the diagnostic “Task Stream” plot on the dashboard. It includes the start,
stop, transfer, and deserialization time of every task for a particular duration.

Note that the task stream diagnostic does not run by default. You may wish to call this function once
before you start work to ensure that things start recording, and then again after you have completed.

Parameters

start: Number or string  When you want to start recording If a number it should be the
result of calling time() If a string then it should be a time difference before now, like
‘60s’ or ‘500 ms’

stop: Number or string  When you want to stop recording

count: int  The number of desired records, ignored if both start and stop are specified

plot: boolean, str  If true then also return a Bokeh figure If plot == ‘save’ then save the
figure to a file

filename: str (optional)  The filename to save to if you set plot='save'

Returns

L: List[Dict]

See also:

get_task_stream  a context manager version of this method

Examples

```python
>>> client.get_task_stream()  # prime plugin if not already connected
>>> x.compute()  # do some work
>>> client.get_task_stream()
```
Pass the `plot=True` or `plot='save'` keywords to get back a Bokeh figure.

```python
>>> data, figure = client.get_task_stream(plot='save', filename='myfile.html')
```

Alternatively consider the context manager.

```python
>>> from dask.distributed import get_task_stream
>>> with get_task_stream() as ts:
...     x.compute()
>>> ts.data
[...]
```

**get_versions** *(check=False, packages=[])*

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
- `packages` [List[str]] Extra package names to check

**Examples**

```python
>>> c.get_versions()  # doctest: +SKIP
```

```python
>>> c.get_versions(packages=['sklearn', 'geopandas'])  # doctest: +SKIP
```

**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

This returns the keys of the data that are held in each worker’s memory.

**Parameters**

- `workers`: list (optional) A list of worker addresses, defaults to all
See also:

\texttt{Client.who\_has, Client.ncores, Client.processing}

**Examples**

```python
>>> x, y, z = c.map(inc, [1, 2, 3])  # doctest: +SKIP
>>> wait([x, y, z])  # doctest: +SKIP
>>> c.has_what()  # doctest: +SKIP
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
 'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
 'inc-le297fc27658d7b67b3a758f16bcf47a']}
```

\texttt{list\_datasets(**kwargs)}

List named datasets available on the scheduler

See also:

\texttt{Client.publish\_dataset, Client.get\_dataset}

\texttt{map(func, *iterables, **kwargs)}

Map a function on a sequence of arguments

Arguments can be normal objects or Futures

**Parameters**

- \texttt{func}: callable
- \texttt{iterables}: Iterables, Iterators, or Queues
- \texttt{key}: str, list Prefix for task names if string. Explicit names if list.
- \texttt{pure}: bool (defaults to True) Whether or not the function is pure. Set \texttt{pure=False} for impure functions like \texttt{np.random.random}.
- \texttt{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)
- \texttt{retries}: int (default to 0) Number of allowed automatic retries if a task fails
- \texttt{priority}: Number Optional prioritization of task. Zero is default. Higher priorities take precedence
- \texttt{fifo\_timeout}: str timedelta (default ‘100ms’) Allowed amount of time between calls to consider the same priority
- \texttt{**kwargs}: dict Extra keywords to send to the function. Large values will be included explicitly in the task graph.

**Returns**

List, iterator, or Queue of futures, depending on the type of the inputs.

See also:

\texttt{Client.submit} Submit a single function
Examples

```python
>>> L = client.map(func, sequence)  # doctest: +SKIP
```

**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`

```python
>>> x, y, z = c.map(inc, [1, 2, 3])  # doctest: +SKIP
>>> c.nbytes(summary=False)  # doctest: +SKIP
{'inc-1c8dd6be1c21646c71f76c16d09304ea': 28,
 'inc-1e297fc27658d7b67b3a758f16bcf47a': 28,
 'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b': 28}
```

```python
>>> c.nbytes(summary=True)  # doctest: +SKIP
{'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`

```python
>>> c.ncores()  # doctest: +SKIP
{'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

4.11. Futures
Examples

```python
>>> len(x.__dask_graph__())  # x is a dask collection with 100 tasks
100
```

```python
>>> set(client.futures).intersection(x.__dask_graph__())  # some overlap
exists  # doctest: +SKIP
10
```

```python
>>> x = client.normalize_collection(x)  # doctest: +SKIP
>>> len(x.__dask_graph__())  # smaller computational graph  # doctest: +SKIP
20
```

**persist** *(collections, optimize_graph=True, workers=None, allow_other_workers=None, resources=None, retries=None, priority=0, fifo_timeout='60s', actors=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

- **priority**: Number  
Optional prioritization of task. Zero is default. Higher priorities take precedence

- **fifo_timeout**: timedelta str (defaults to '60s')  
Allowed amount of time between calls to consider the same priority

- **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**

```python
>>> xx = client.persist(x)  # doctest: +SKIP
>>> xx, yy = client.persist([x, y])  # doctest: +SKIP
```

**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.who_has, Client.has_what, Client.ncores

Examples

```python
>>> x, y, z = c.map(inc, [1, 2, 3])  # doctest: +SKIP
>>> c.processing()  # doctest: +SKIP
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
 'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
 'inc-1e297fc27658d7b67b3a758f16bcf47a']}
```

profile (key=None, start=None, stop=None, workers=None, merge_workers=True, plot=False, filename=None)

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

plot: boolean or string  Whether or not to return a plot object

filename: str  Filename to save the plot

Examples

```python
>>> client.profile()  # call on collections
>>> client.profile(filename='dask-profile.html')  # save to html file
```

publish_dataset (*args, **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

args  [list of objects to publish as name]

name  [optional name of the dataset to publish]

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:

```python
>>> df = dd.read_csv('s3://...')  # doctest: +SKIP
>>> df = c.persist(df)  # doctest: +SKIP
>>> c.publish_dataset(my_dataset=df)  # doctest: +SKIP
```

Alternative invocation >>> c.publish_dataset(df, name='my_dataset')

Receiving client:

```python
>>> c.list_datasets()  # doctest: +SKIP
['my_dataset']
>>> df2 = c.get_dataset('my_dataset')  # doctest: +SKIP
```

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

register_worker_callbacks (setup=None)

Registers a setup callback function for all current and future workers.

This registers a new setup function for workers in this cluster. The function will run immediately on all currently connected workers. It will also be run upon connection by any workers that are added in the future. Multiple setup functions can be registered - these will be called in the order they were added.

If the function takes an input argument named dask_worker then that variable will be populated with the worker itself.

Parameters

- **setup** [callable(dask_worker: Worker) -> None] Function to register and run on 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

```python
>>> x = c.submit(func, *args)  # doctest: +SKIP
>>> c.replicate([x])  # send to all workers  # doctest: +SKIP
>>> c.replicate([x], n=3)  # send to three workers  # doctest: +SKIP
>>> c.replicate([x], workers=['alice', 'bob'])  # send to specific #  # doctest: +SKIP
>>> c.replicate([x], n=1, workers=['alice', 'bob'])  # send to one of specific workers  # doctest: +SKIP
>>> c.replicate([x], n=1)  # reduce replications  # doctest: +SKIP
```

restart(**kwargs)

Restart the distributed network

This kills all active work, deletes all data on the network, and restarts the worker processes.

retire_workers(workers=None, close_workers=True, **kwargs)

Retire certain workers on the scheduler

See dask.distributed.Scheduler.retire_workers for the full docstring.

See also:

dask.distributed.Scheduler.retire_workers

Examples

You can get information about active workers using the following:

```python
>>> workers = client.scheduler_info()['workers']
```

From that list you may want to select some workers to close

```python
>>> client.retire_workers(workers=['tcp://address:port', ...])
```

retry(futures, asynchronous=None)

Retry failed futures

Parameters

  futures: list of Futures

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 and 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.
- `wait`: boolean (optional) If the function is asynchronous whether or not to wait until that function finishes.

**Examples**

```python
>>> c.run(os.getpid)  # doctest: +SKIP
{'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.

```python
>>> c.run(os.getpid, workers=['192.168.0.100:9000', ...
 '192.168.0.101:9000'])  # doctest: +SKIP
{'192.168.0.100:9000': 1234,
 '192.168.0.101:9000': 4321}
```

```python
>>> def get_status(dask_worker):
...     return dask_worker.status
```

```python
>>> c.run(get_status)  # doctest: +SKIP
{'192.168.0.100:9000': 'running',
 '192.168.0.101:9000': 'running'}
```

Run asynchronous functions in the background:

```python
>>> async def print_state(dask_worker):  # doctest: +SKIP
...     while True:
...         print(dask_worker.status)
...         await gen.sleep(1)
```

```python
>>> c.run(print_state, wait=False)  # doctest: +SKIP
```

**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

```python
>>> def get_number_of_tasks(dask_scheduler=None):
...     return len(dask_scheduler.tasks)
```

```python
>>> client.run_on_scheduler(get_number_of_tasks)  # doctest: +SKIP
100
```

Run asynchronous functions in the background:

```python
>>> async def print_state(dask_scheduler):  # doctest: +SKIP
...     while True:
...         print(dask_scheduler.status)
...         await gen.sleep(1)
```

```python
>>> c.run(print_state, wait=False)  # doctest: +SKIP
```

scatter (data, workers=None, broadcast=False, direct=None, hash=True, maxsize=0, timeout='__no_default__', 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) Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary. This can also be set when creating the Client.

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

```python
>>> c = Client('127.0.0.1:8787')  # doctest: +SKIP
>>> c.scatter(1)  # doctest: +SKIP
<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>

>>> c.scatter([1, 2, 3])  # doctest: +SKIP
[<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>,
 <Future: status: finished, key: 58e78e1b34eb49a68c6554815d1b158>,
 <Future: status: finished, key: d3395e15f605bc35ab1bac6341a285e2>]

>>> c.scatter({'x': 1, 'y': 2, 'z': 3})  # doctest: +SKIP
{'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

```python
>>> c.scatter([1, 2, 3], workers=[('hostname', 8788)])  # doctest: +SKIP

Handle streaming sequences of data with iterators or queues

```python
>>> seq = c.scatter(iter([1, 2, 3]))  # doctest: +SKIP
>>> next(seq)  # doctest: +SKIP
<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>,

Broadcast data to all workers

```python
>>> [future] = c.scatter([element], broadcast=True)  # doctest: +SKIP

Send scattered data to parallelized function using client futures interface

```python
>>> data = c.scatter(data, broadcast=True)  # doctest: +SKIP
>>> res = [c.submit(func, data, i) for i in range(100)]

scheduler_info(**kwargs)
Basic information about the workers in the cluster

Examples

```python
>>> c.scheduler_info()  # doctest: +SKIP
{'id': '2de2b6da-69ee-11e6-ab6a-e82aea155996',
 'services': {},
 'type': 'Scheduler',

(continues on next page)
'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:

```python
>>> client.set_metadata(['a', 'b', 'c'], 123)
```

Then this is the same as setting

```python
>>> scheduler.task_metadata['a']['b']['c'] = 123
```

The lower level dictionaries will be created on demand.

See also:

get_metadata

Examples

```python
>>> client.set_metadata('x', 123)  # doctest: +SKIP
>>> client.get_metadata('x')  # doctest: +SKIP
123
```

```python
>>> client.set_metadata(['x', 'y'], 123)  # doctest: +SKIP
>>> client.get_metadata('x')  # doctest: +SKIP
{'y': 123}
```

```python
>>> client.set_metadata(['x', 'w', 'z'], 456)  # doctest: +SKIP
>>> client.get_metadata('x')  # doctest: +SKIP
{'y': 123, 'w': {'z': 456}}
```

```python
>>> client.get_metadata(['x', 'w'])  # doctest: +SKIP
{'z': 456}
```

shutdown(*args, **kwargs)

Deprecated, see close instead

This was deprecated because “shutdown” was sometimes confusingly thought to refer to the cluster rather than the client

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**

```python
>>> c.start_ipython_scheduler() # doctest: +SKIP
%scheduler scheduler.processing # doctest: +SKIP
{'127.0.0.1:3595': {'inc-1', 'inc-2'},
 '127.0.0.1:53589': {'inc-2', 'add-5'}}
```

```python
>>> c.start_ipython_scheduler(qtconsole=True) # doctest: +SKIP
```

**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 asterisk then expand asterisk 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

```python
>>> info = c.start_ipython_workers()  # doctest: +SKIP
>>> %remote info['192.168.1.101:5752'] worker.data  # doctest: +SKIP
{'x': 1, 'y': 100}

>>> c.start_ipython_workers('192.168.1.101:5752', magic_names='w')  # doctest: +SKIP
>>> %w worker.data  # doctest: +SKIP
{'x': 1, 'y': 100}

>>> c.start_ipython_workers('192.168.1.101:5752', qtconsole=True)  # doctest: +SKIP
>>> %w worker.data  # doctest: +SKIP
{'x': 1, 'y': 100}
```

Add asterix * in magic names to add one magic per worker

```python
>>> c.start_ipython_workers('192.168.1.101:5752', magic_names='w_*')  # doctest: +SKIP
>>> %w_0 worker.data  # doctest: +SKIP
{'x': 1, 'y': 100}
>>> %w_1 worker.data  # doctest: +SKIP
{'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
  - **priority**: Number Optional prioritization of task. Zero is default. Higher priorities take precedence
  - **fifo_timeout**: str timedelta (default ‘100ms’) Allowed amount of time between calls to consider the same priority

**Returns**

Future

See also:

`Client.map` Submit on many arguments at once
Examples

```python
>>> c = client.submit(add, a, b) # doctest: +SKIP
```

**unpublish_dataset** *(name, **kwargs)*

Remove named datasets from scheduler

See also:

`Client.publish_dataset`

Examples

```python
>>> c.list_datasets() # doctest: +SKIP
['my_dataset']
```

```python
c.unpublish_datasets('my_dataset') # doctest: +SKIP
c.list_datasets() # doctest: +SKIP
[]
```

**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, .egg or .zip files will be importable.

Parameters

- **filename**: string  Filename of .py, .egg or .zip file to send to workers

Examples

```python
>>> client.upload_file('mylibrary.egg') # doctest: +SKIP
>>> from mylibrary import myfunc # doctest: +SKIP
>>> L = c.map(myfunc, seq) # doctest: +SKIP

```

**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

```python
>>> x, y, z = c.map(inc, [1, 2, 3]) # doctest: +SKIP
>>> wait([x, y, z]) # doctest: +SKIP
>>> c.who_has() # doctest: +SKIP
{'inc-1c8dd6be1c21646c71f76c16d09304ea': ['192.168.1.141:46784'],
'inc-1e297fc27658d7b67b3a758f16bcf47a': ['192.168.1.141:46784'],
'inc-fd65c238a7ea60f6a01bf4c8a5f4f44b': ['192.168.1.141:46784']}
```
>>> c.who_has([x, y])  # doctest: +SKIP
{'inc-1c8dd6be1c21646c71f76c16d09304ea': ['192.168.1.141:46784'],
 'inc-1e297fc27658d7b67b3a758f16bcf47a': ['192.168.1.141:46784']}

write_scheduler_file (scheduler_file)
Write the scheduler information to a json file.
This facilitates easy sharing of scheduler information using a file system. The scheduler file can be used to instantiate a second Client using the same scheduler.

Parameters

scheduler_file: str  Path to a write the scheduler file.

Examples

>>> client = Client()  # doctest: +SKIP
>>> client.write_scheduler_file('scheduler.json')  # doctest: +SKIP
# connect to previous client's scheduler
>>> client2 = Client(scheduler_file='scheduler.json')  # doctest: +SKIP

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)  # doctest: +SKIP

We can track the progress and results of a future

>>> my_future  # doctest: +SKIP
<Future: status: finished, key: add-8f6e709446674bad78ea8aeecfe188e>

We can get the result or the exception and traceback from the future

>>> my_future.result()  # doctest: +SKIP

add_done_callback (fn)
Call callback on future when callback has finished
The callback \texttt{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.

\begin{verbatim}
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?

distance (timeout=None, **kwargs)
    Return the exception of a failed task
    If \texttt{timeout} seconds are elapsed before returning, a \texttt{dask.distributed.TimeoutError} is raised.
    See also:
    Future.traceback

result (timeout=None)
    Wait until computation completes, gather result to local process.
    If \texttt{timeout} seconds are elapsed before returning, a \texttt{dask.distributed.TimeoutError} is raised.

retry(**kwargs)
    Retry this future if it has failed
    See also:
    Client.retry

traceback (timeout=None, **kwargs)
    Return the traceback of a failed task
    This returns a traceback object. You can inspect this object using the \texttt{traceback} module. Alternatively if you call \texttt{future.result()} this traceback will accompany the raised exception.
    If \texttt{timeout} seconds are elapsed before returning, a \texttt{dask.distributed.TimeoutError} is raised.
    See also:
    Future.exception
\end{verbatim}

\section*{Examples}

\begin{verbatim}
>>> import traceback  # doctest: +SKIP
>>> tb = future.traceback()  # doctest: +SKIP
>>> traceback.format_tb(tb)  # doctest: +SKIP
[...]
\end{verbatim}

\begin{verbatim}
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.
\end{verbatim}
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**

```python
>>> from dask.distributed import Client, Queue  # doctest: +SKIP
>>> client = Client()  # doctest: +SKIP
>>> queue = Queue('x')  # doctest: +SKIP
>>> future = client.submit(f, x)  # doctest: +SKIP
>>> queue.put(future)  # doctest: +SKIP
```

**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

```python
>>> from dask.distributed import Client, Variable # doctest: +SKIP
>>> client = Client() # doctest: +SKIP
>>> x = Variable('x') # doctest: +SKIP
>>> x.set(123) # doctest: +SKIP
>>> x.get() # doctest: +SKIP
123
>>> future = client.submit(f, x) # doctest: +SKIP
>>> x.set(future) # doctest: +SKIP
```

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

class distributed.Lock (name=None, client=None)
Distributed Centralized Lock

Parameters

name: string Name of the lock to acquire. Choosing the same name allows two disconnected processes to coordinate a lock.

Examples

```python
>>> lock = Lock('x') # doctest: +SKIP
>>> lock.acquire(timeout=1) # doctest: +SKIP
>>> # do things with protected resource
>>> lock.release() # doctest: +SKIP
```

acquire (blocking=True, timeout=None)
Acquire the lock

Parameters

blocking [bool, optional] If false, don’t wait on the lock in the scheduler at all.

timeout [number, optional] Seconds to wait on the lock in the scheduler. This does not include local coroutine time, network transfer time, etc. It is forbidden to specify a timeout when blocking is false.

Returns

True or False whether or not it sucessfully acquired the lock
Examples

```python
>>> lock = Lock('x')  # doctest: +SKIP
>>> lock.acquire(timeout=1)  # doctest: +SKIP
```

**release()**

Release the lock if already acquired

```python
def class distributed.Pub(name, worker=None, client=None):
    # Publish data with Publish-Subscribe pattern
    This allows clients and workers to directly communicate data between each other with a typical Publish-Subscribe pattern. This involves two components,
    Pub objects, into which we put data:
    ```
    >>> pub = Pub('my-topic')
    >>> pub.put(123)
    ```
    And Sub objects, from which we collect data:
    ```
    >>> sub = Sub('my-topic')
    >>> sub.get()
    123
    ```
    Many Pub and Sub objects can exist for the same topic. All data sent from any Pub will be sent to all Sub objects on that topic that are currently connected. Pub's and Sub's find each other using the scheduler, but they communicate directly with each other without coordination from the scheduler.

    Pubs and Subs use the central scheduler to find each other, but not to mediate the communication. This means that there is very little additional latency or overhead, and they are appropriate for very frequent data transfers. For context, most data transfer first checks with the scheduler to find which workers should participate, and then does direct worker-to-worker transfers. This checking in with the scheduler provides some stability guarantees, but also adds in a few extra network hops. PubSub doesn't do this, and so is faster, but also can easily drop messages if Pubs or Subs disappear without notice.

    When using a Pub or Sub from a Client all communications will be routed through the scheduler. This can cause some performance degradation. Pubs an Subs only operate at top-speed when they are both on workers.

    **Parameters**
    ```
    name: object (msgpack serializable)  The name of the group of Pubs and Subs on which to participate
    ```

    **See also:**
    ```
    Sub
    ```

    **Examples**
    ```
    >>> pub = Pub('my-topic')
    >>> sub = Sub('my-topic')
    >>> pub.put([1, 2, 3])
    >>> sub.get()
    [1, 2, 3]
    ```
    You can also use sub within a for loop:
or an async for loop

```python
>>> async for msg in sub:  # doctest: +SKIP
...     print(msg)
```

Similarly the `.get` method will return an awaitable if used by an async client or within the IOLoop thread of a worker

```python
>>> await sub.get()  # doctest: +SKIP
```

You can see the set of connected worker subscribers by looking at the `.subscribers` attribute:

```python
>>> pub.subscribers
{'tcp://...': {},
 'tcp://...': {}}
```

**put** *(msg)*

Publish a message to all subscribers of this topic

**class** `distributed.Sub` *(name, worker=None, client=None)*

Subscribe to a Publish/Subscribe topic

See also:

- `Pub` for full docstring
- **get** *(timeout=None)*
  Get a single message
- **next** *(timeout=None)*
  Get a single message

### 4.12 Best Practices

It is easy to get started with Dask’s APIs, but using them well requires some experience. This page contains suggestions for best practices, and includes solutions to common problems.

This document specifically focuses on best practices that are shared among all of the Dask APIs. Readers may first want to investigate one of the API-specific Best Practices documents first.

- **Arrays**
- **DataFrames**
- **Delayed**

#### 4.12.1 Start Small

Parallelism brings extra complexity and overhead. Sometimes it’s necessary for larger problems, but often it’s not. Before adding a parallel computing system like Dask to your workload you may want to first try some alternatives:
• **Use better algorithms or data structures:** NumPy, Pandas, Scikit-Learn may have faster functions for what you’re trying to do. It may be worth consulting with an expert or reading through their docs again to find a better pre-built algorithm.

• **Better file formats:** Efficient binary formats that support random access can often help you manage larger-than-memory datasets efficiently and simply. See the *Store Data Efficiently* section below.

• **Compiled code:** Compiling your Python code with Numba or Cython might make parallelism unnecessary. Or you might use the multi-core parallelism available within those libraries.

• **Sampling:** Even if you have a lot of data, there might not be much advantage from using all of it. By sampling intelligently you might be able to derive the same insight from a much more manageable subset.

• **Profile:** If you’re trying to speed up slow code it’s important that you first understand why it is slow. Modest time investments in profiling your code can help you to identify what is slowing you down. This information can help you make better decisions about if parallelism is likely to help, or if other approaches are likely to be more effective.

### 4.12.2 Use The Dashboard

Dask’s dashboard helps you to understand the state of your workers. This information can help to guide you to efficient solutions. In parallel and distributed computing there are new costs to be aware of and so your old intuition may no longer be true. Working with the dashboard can help you relearn about what is fast and slow and how to deal with it.

See *Documentation on Dask’s dashboard* for more information.

### 4.12.3 Avoid Very Large Partitions

Your chunks of data should be small enough so that many of them fit in a worker’s available memory at once. You often control this when you select partition size in Dask DataFrame or chunk size in Dask Array.

Dask will likely manipulate as many chunks in parallel on one machine as you have cores on that machine. So if you have 1 GB chunks and ten cores, then Dask is likely to use at least 10 GB of memory. Additionally, it’s common for Dask to have 2-3 times as many chunks available to work on so that it always has something to work on.

If you have a machine with 100 GB and 10 cores, then you might want to choose chunks in the 1GB range. You have space for ten chunks per core which gives Dask a healthy margin, without having tasks that are too small.

Note that you also want to avoid chunk sizes that are too small. See the next section for details.

### 4.12.4 Avoid Very Large Graphs

Dask workloads are composed of *tasks*. A task is a Python function, like `np.sum` applied onto a Python object, like a Pandas dataframe or NumPy array. If you are working with Dask collections with many partitions, then every operation you do, like `x + 1` likely generates many tasks, at least as many as partitions in your collection.

Every task comes with some overhead. This is somewhere between 200us and 1ms. If you have a computation with thousands of tasks this is fine, there will be about a second of overhead, and that may not trouble you.

However when you have very large graphs with millions of tasks then this may become troublesome, both because overhead is now in the 10 minutes to hours range, and also because the overhead of dealing with such a large graph can start to overwhelm the scheduler.

There are a few things you can do to address this:

• Build smaller graphs. You can do this by...
- **Increasing your chunk size:** If you have a 1000 GB of data and are using 10 MB chunks, then you have 100,000 partitions. Every operation on such a collection will generate at least 100,000 tasks.

  However if you increase your chunksize to 1 GB or even a few GB then you reduce the overhead by orders of magnitude. This requires that your workers have much more than 1 GB of memory, but that’s typical for larger workloads.

- **Fusing operations together:** Dask will do a bit of this on its own, but you can help it. If you have a very complex operation with dozens of sub-operations, maybe you can pack that into a single Python function and use a function like `da.map_blocks` or `dd.map_partitions`.

  In general, the more administrative work you can move into your functions the better. That way the Dask scheduler doesn’t need to think about all of the fine-grained operations.

- **Breaking up your computation:** For very large workloads you may also want to try sending smaller chunks to Dask at a time. For example if you’re processing a petabyte of data but find that Dask is only happy with 100 TB, maybe you can break up your computation into ten pieces and submit them one after the other.

### 4.12.5 Learn Techniques For Customization

The high level Dask collections (array, dataframe, bag) include common operations that follow standard Python APIs from NumPy and Pandas. However, many Python workloads are complex and may require operations that are not included in these high level APIs.

Fortunately, there are many options to support custom workloads:

- All collections have a `map_partitions` or `map_blocks` function, that applies a user provided function across every Pandas dataframe or NumPy array in the collection. Because Dask collections are made up of normal Python objects, it’s often quite easy to map custom functions across partitions of a dataset without much modification.

```python
df.map_partitions(my_custom_func)
```

- More complex `map_*` functions. Sometimes your custom behavior isn’t embarrassingly parallel, but requires more advanced communication. For example maybe you need to communicate a little bit of information from one partition to the next, or maybe you want to build a custom aggregation.

  Dask collections include methods for these as well.

- For even more complex workloads you can convert your collections into individual blocks, and arrange those blocks as you like using Dask Delayed. There is usually a `to_delayed` method on every collection.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map_partitions(func, *args, **kwargs)</code></td>
<td>Apply Python function on each DataFrame partition.</td>
</tr>
<tr>
<td><code>rolling.map_overlap(func, df, before, after, ...)</code></td>
<td>Apply a function to each partition, sharing rows with adjacent partitions.</td>
</tr>
<tr>
<td><code>Aggregation(name, chunk, agg[, finalize])</code></td>
<td>User defined groupby-aggregation.</td>
</tr>
<tr>
<td><code>blockwise(func, out_ind, *args, **kwargs)</code></td>
<td>Tensor operation: Generalized inner and outer products</td>
</tr>
<tr>
<td><code>map_blocks(func, *args, **kwargs)</code></td>
<td>Map a function across all blocks of a dask array.</td>
</tr>
<tr>
<td><code>map_overlap(x, func, depth[, boundary, trim])</code></td>
<td>Map a function over blocks of the array with some overlap</td>
</tr>
<tr>
<td><code>reduction(x, chunk, aggregate[, axis, ...])</code></td>
<td>General version of reductions</td>
</tr>
</tbody>
</table>
4.12.6 Stop Using Dask When No Longer Needed

In many workloads it is common to use Dask to read in a large amount of data, reduce it down, and then iterate on a much smaller amount of data. For this latter stage on smaller data it may make sense to stop using Dask, and start using normal Python again.

```python
df = dd.read_parquet("lots-of-data-*.parquet")
df = df.groupby('name').mean()  # reduce data significantly
df = df.compute()  # continue on with Pandas/NumPy
```

4.12.7 Persist When You Can

Accessing data from RAM is often much faster than accessing it from disk. Once you have your dataset in a clean state that both:

1. Fits in memory
2. Is clean enough that you will want to try many different analyses

Then it is a good time to persist your data in RAM

```python
df = dd.read_parquet("lots-of-data-*.parquet")
df = df.fillna(...)  # clean up things lazily
df = df[df.name == 'Alice']  # get down to a more reasonable size
df = df.persist()  # trigger computation, persist in distributed RAM
```

Note that this is only relevant if you are on a distributed machine (otherwise, as mentioned above, you should probably continue on without Dask).

4.12.8 Store Data Efficiently

As your ability to compute increases you will likely find that data access and I/O take up a larger portion of your total time. Additionally, parallel computing will often add new constraints to how your store your data, particularly around providing random access to blocks of your data that are in line with how you plan to compute on it.

For example …

- For compression you'll probably find that you drop gzip and bz2, and embrace newer systems like lz4, snappy, and Z-Standard that provide better performance and random access.
- For storage formats you may find that you want self-describing formats that are optimized for random access, metadata storage, and binary encoding like Parquet, ORC, Zarr, HDF5, GeoTIFF and so on
- When working on the cloud you may find that some older formats like HDF5 may not work well
- You may want to partition or chunk your data in ways that align well to common queries. In Dask DataFrame this might mean choosing a column to sort by for fast selection and joins. For Dask dataframe this might mean choosing chunk sizes that are aligned with your access patterns and algorithms.

4.13 API

Dask APIs generally follow from upstream APIs:

- The Dask Array API follows the NumPy API
• The Dask DataFrame API follows the Pandas API
• The Dask-ML API follows the Scikit-Learn API and other related machine learning libraries
• The Dask Bag API follows the map/filter/groupby/reduce API common in PySpark, PyToolz, and the Python standard library
• The Dask Delayed API wraps general Python code
• The Real-time Futures API follows the concurrent.futures API from the standard library.

Additionally, Dask has its own functions to start computations, persist data in memory, check progress, and so forth that complement the APIs above. These more general Dask functions are described below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compute(*args, **kwargs)</code></td>
<td>Compute several dask collections at once.</td>
</tr>
<tr>
<td><code>is_dask_collection(x)</code></td>
<td>Returns <code>True</code> if <code>x</code> is a dask collection</td>
</tr>
<tr>
<td><code>optimize(*args, **kwargs)</code></td>
<td>Optimize several dask collections at once.</td>
</tr>
<tr>
<td><code>persist(*args, **kwargs)</code></td>
<td>Persist multiple Dask collections into memory</td>
</tr>
<tr>
<td><code>visualize(*args, **kwargs)</code></td>
<td>Visualize several dask graphs at once.</td>
</tr>
</tbody>
</table>

These functions work with any scheduler. More advanced operations are available when using the newer scheduler and starting a `dask.distributed.Client` (which, despite its name, runs nicely on a single machine). This API provides the ability to submit, cancel, and track work asynchronously, and includes many functions for complex inter-task workflows. These are not necessary for normal operation, but can be useful for real-time or advanced operation.

This more advanced API is available in the Dask distributed documentation:

```python
import dask.array as da

a = da.arange(10, chunks=2).sum()
b = da.arange(10, chunks=2).mean()
compute(a, b)
```

Examples
By default, dask objects inside python collections will also be computed:

```python
>>> compute({'a': a, 'b': b, 'c': 1}) # doctest: +SKIP
({'a': 45, 'b': 4.5, 'c': 1},)
```

**dask.is_dask_collection(x)**

Returns True if x is a dask collection.

**dask.optimize(*args, **kwargs)**

Optimize several dask collections at once.

Returns equivalent dask collections that all share the same merged and optimized underlying graph. This can be useful if converting multiple collections to delayed objects, or to manually apply the optimizations at strategic points.

Note that in most cases you shouldn’t need to call this method directly.

**Parameters**

- **args** [objects] Any number of objects. If a dask object, its graph is optimized and merged with all those of all other dask objects before returning an equivalent dask collection. Non-dask arguments are passed through unchanged.

- **traverse** [bool, optional] By default dask traverses builtin python collections looking for dask objects passed to optimize. For large collections this can be expensive. If none of the arguments contain any dask objects, set `traverse=False` to avoid doing this traversal.

- **optimizations** [list of callables, optional] Additional optimization passes to perform.

- **kwargs** Extra keyword arguments to forward to the optimization passes.

**Examples**

```python
>>> import dask.array as da
>>> a = da.arange(10, chunks=2).sum()
>>> b = da.arange(10, chunks=2).mean()
>>> a2, b2 = optimize(a, b)

>>> a2.compute() == a.compute()
True
>>> b2.compute() == b.compute()
True
```

**dask.persist(*args, **kwargs)**

Persist multiple Dask collections into memory.

This turns lazy Dask collections into Dask collections with the same metadata, but now with their results fully computed or actively computing in the background.

For example a lazy dask.array built up from many lazy calls will now be a dask.array of the same shape, dtype, chunks, etc., but now with all of those previously lazy tasks either computed in memory as many small numpy.array (in the single-machine case) or asynchronously running in the background on a cluster (in the distributed case).

This function operates differently if a dask.distributed.Client exists and is connected to a distributed scheduler. In this case this function will return as soon as the task graph has been submitted to the cluster, but before the computations have completed. Computations will continue asynchronously in the background. When using this function with the single machine scheduler it blocks until the computations have finished.

When using Dask on a single machine you should ensure that the dataset fits entirely within memory.
Parameters

*args: Dask collections

scheduler [string, optional] Which scheduler to use like “threads”, “synchronous” or “processes”. If not provided, the default is to check the global settings first, and then fall back to the collection defaults.

traverse [bool, optional] By default dask traverses builtin python collections looking for dask objects passed to persist. For large collections this can be expensive. If none of the arguments contain any dask objects, set traverse=False to avoid doing this traversal.

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 function.

Returns

New dask collections backed by in-memory data

Examples

>>> df = dd.read_csv('/path/to/*.csv') # doctest: +SKIP
>>> df = df[df.name == 'Alice'] # doctest: +SKIP
>>> df['in-debt'] = df.balance < 0 # doctest: +SKIP
>>> df = df.persist() # triggers computation # doctest: +SKIP

>>> df.value().min() # future computations are now fast # doctest: +SKIP
-10
>>> df.value().max() # doctest: +SKIP
100

>>> from dask import persist # use persist function on multiple collections
>>> a, b = persist(a, b) # doctest: +SKIP

dask.visualize(*args, **kwargs)
Visualize several dask graphs at once.

Requires graphviz to be installed. All options that are not the dask graph(s) should be passed as keyword arguments.

Parameters

dsk [dict(s) or collection(s)] The dask graph(s) to visualize.

filename [str or None, optional] The name (without an extension) of the file to write to disk.


optimize_graph [bool, optional] If True, the graph is optimized before rendering. Otherwise, the graph is displayed as is. Default is False.

color: {None, ‘order’}, optional Options to color nodes. Provide cmap= keyword for additional colormap

**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.dot.dot_graph

Notes

For more information on optimization see here:

https://docs.dask.org/en/latest/optimize.html

Examples

>>> x.visualize(filename='dask.pdf')  # doctest: +SKIP
>>> x.visualize(filename='dask.pdf', color='order')  # doctest: +SKIP

Scheduling

Schedulers execute task graphs. Dask currently has two main schedulers: one for local processing using threads or processes; and one for distributed memory clusters.

- Scheduling
- Distributed Scheduling

4.14 Scheduling

All of the large-scale Dask collections like Dask Array, Dask DataFrame, and Dask Bag and the fine-grained APIs like delayed and futures generate task graphs where each node in the graph is a normal Python function and edges between nodes are normal Python objects that are created by one task as outputs and used as inputs in another task. After Dask generates these task graphs, it needs to execute them on parallel hardware. This is the job of a task scheduler. Different task schedulers exist, and each will consume a task graph and compute the same result, but with different performance characteristics.

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, although 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.

For different computations you may find better performance with particular scheduler settings. This document helps you understand how to choose between and configure different schedulers, and provides guidelines on when one might be more appropriate.

### 4.14.1 Local Threads

```python
import dask
dask.config.set(scheduler='threads')  # overwrite default with threaded scheduler
```

The threaded scheduler executes computations with a local `multiprocessing.pool.ThreadPool`. It is lightweight and requires no setup. It introduces very little task overhead (around 50us per task) and, because everything occurs in the same process, it incurs no costs to transfer data between tasks. However, due to Python’s Global Interpreter Lock (GIL), this scheduler only provides parallelism when your computation is dominated by non-Python code, such as is the case when operating on numeric data in NumPy arrays, Pandas DataFrames, or using any of the other C/C++/Cython based projects in the ecosystem.

The threaded scheduler is the default choice for *Dask Array*, *Dask DataFrame*, and *Dask Delayed*. However, if your computation is dominated by processing pure Python objects like strings, dicts, or lists, then you may want to try one of the process-based schedulers below (we currently recommend the distributed scheduler on a local machine).

### 4.14.2 Local Processes

**Note:** The distributed scheduler described a couple sections below is often a better choice today. We encourage readers to continue reading after this section.

```python
import dask.multiprocessing
dask.config.set(scheduler='processes')  # overwrite default with multiprocessing scheduler
```

The multiprocessing scheduler executes computations with a local `multiprocessing.Pool`. It is lightweight to use and requires no setup. Every task and all of its dependencies are shipped to a local process, executed, and then their result is shipped back to the main process. This means that it is able to bypass issues with the GIL and provide parallelism even on computations that are dominated by pure Python code, such as those that process strings, dicts, and lists.

However, moving data to remote processes and back can introduce performance penalties, particularly when the data being transferred between processes is large. The multiprocessing scheduler is an excellent choice when workflows are relatively linear, and so does not involve significant inter-task data transfer as well as when inputs and outputs are both small, like filenames and counts.

This is common in basic data ingestion workloads, such as those are common in *Dask Bag*, where the multiprocessing scheduler is the default:

```python
>>> import dask.bag as db
>>> db.read_text('*.json').map(json.loads).pluck('name').frequencies().compute()
{'alice': 100, 'bob': 200, 'charlie': 300}
```

For more complex workloads, where large intermediate results may be depended upon by multiple downstream tasks, we generally recommend the use of the distributed scheduler on a local machine. The distributed scheduler is more intelligent about moving around large intermediate results.
4.14.3 Single Thread

```python
import dask
dask.config.set(scheduler='synchronous')  # overwrite default with single-threaded
```

The single-threaded synchronous scheduler executes all computations in the local thread with no parallelism at all. This is particularly valuable for debugging and profiling, which are more difficult when using threads or processes.

For example, when using IPython or Jupyter notebooks, the `%debug`, `%pdb`, or `%prun` magics will not work well when using the parallel Dask schedulers (they were not designed to be used in a parallel computing context). However, if you run into an exception and want to step into the debugger, you may wish to rerun your computation under the single-threaded scheduler where these tools will function properly.

4.14.4 Dask Distributed (local)

```python
from dask.distributed import Client
client = Client()
# or
client = Client(processes=False)
```

The Dask distributed scheduler can either be setup on a cluster or run locally on a personal machine. Despite having the name “distributed”, it is often pragmatic on local machines for a few 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 read more about using the Dask distributed scheduler on a single machine in these docs.

4.14.5 Dask Distributed (Cluster)

You can also run Dask on a distributed cluster. There are a variety of ways to set this up depending on your cluster. We recommend referring to the setup documentation for more information.

4.14.6 Configuration

You can configure the global default scheduler by using the `dask.config.set(scheduler...)` command. This can be done globally:

```python
dask.config.set(scheduler='threads')
x.compute()
```

or as a context manager:

```python
with dask.config.set(scheduler='threads'):
    x.compute()
```

or within a single compute call:
Additionally some of the scheduler support other keyword arguments. For example, the pool-based single-machine scheduler allows you to provide custom pools or specify the desired number of workers:

```python
from multiprocessing.pool import ThreadPool
with dask.config.set(pool=ThreadPool(4)):
    ...
with dask.config.set(num_workers=4):
    ...
```

4.15 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:


Diagnosing Performance

Parallel code can be tricky to debug and profile. Dask provides several tools to help make debugging and profiling graph execution easier.

- Understanding Performance
- Visualize task graphs
- Diagnostics (local)
- Diagnostics (distributed)
- Debugging

4.16 Understanding Performance

The first step in making computations run quickly is to understand the costs involved. In Python we often rely on tools like the CProfile module, %%prun IPython magic, VMProf, or snakeviz to understand the costs associated with our code. However, few of these tools work well on multi-threaded or multi-process code, and fewer still on computations distributed among many machines. We also have new costs like data transfer, serialization, task scheduling overhead, and more that we may not be accustomed to tracking.

Fortunately, the Dask schedulers come with diagnostics to help you understand the performance characteristics of your computations. By using these diagnostics and with some thought, we can often identify the slow parts of troublesome computations.

The single-machine and distributed schedulers come with different diagnostic tools. These tools are deeply integrated into each scheduler, so a tool designed for one will not transfer over to the other.

These pages provide four options for profiling parallel code:

1. Visualize task graphs
2. Single threaded scheduler and a normal Python profiler
3. Diagnostics for the single-machine scheduler
4. Dask distributed dashboard
Additionally, if you are interested in understanding the various phases where slowdown can occur, you may wish to read the following:

- Phases of computation

### 4.17 Visualize task graphs

```python
visualize(*args, **kwargs) Visualize several dask graphs at once.
```

Before executing your computation you might consider visualizing the underlying task graph. By looking at the interconnectedness of tasks you can learn more about potential bottlenecks where parallelism may not be possible, or areas where many tasks depend on each other, which may cause a great deal of communication.

The `.visualize` method and `dask.visualize` function work exactly like the `.compute` method and `dask.compute` function, except that rather than computing the result, they produce an image of the task graph.

By default the task graph is rendered from top to bottom. In the case that you prefer to visualize it from left to right, pass `rankdir="LR"` as a keyword argument to `.visualize`.

```python
import dask.array as da
x = da.ones((15, 15), chunks=(5, 5))
y = x + x.T
# y.compute()
y.visualize(filename='transpose.svg')
```

Note that the `visualize` function is powered by the GraphViz system library. This library has a few considerations:

1. You must install both the graphviz system library (with tools like apt-get, yum, or brew) and the graphviz Python library. If you use Conda then you need to install `python-graphviz`, which will bring along the `graphviz` system library as a dependency.

2. Graphviz takes a while on graphs larger than about 100 nodes. For large computations you might have to simplify your computation a bit for the `visualize` method to work well.

### 4.18 Diagnostics (local)

Profiling parallel code can be challenging, but `dask.diagnostics` provides functionality to aid in profiling and inspecting execution with the local task scheduler.

This page describes the following few built-in options:

1. ProgressBar
2. Profiler
3. ResourceProfiler
4. CacheProfiler

Furthermore, this page then provides instructions on how to build your own custom diagnostic.
4.18.1 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 give 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:

```python
>>> 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:

```python
>>> pbar = ProgressBar()
>>> pbar.register()

>>> out = res.compute()
[########################################] | 100% Completed | 17.1 s
```

To unregister from the global callbacks, call the `unregister` method:

```python
>>> pbar.unregister()
```

4.18.2 Profiler

Dask provides a few tools for profiling execution. As with the `ProgressBar`, they each can be used as context managers or registered globally.

The `Profiler` class is used to profile Dask’s 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

4.18.3 ResourceProfiler

The `ResourceProfiler` class is used to profile Dask’s 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:

```python
>>> from dask.diagnostics import ResourceProfiler
>>> rprof = ResourceProfiler(dt=0.5)
```

### 4.18.4 CacheProfiler

`CacheProfiler(metric, metric_name)` A profiler for Dask execution at the scheduler cache level.

The `CacheProfiler` class is used to profile Dask’s 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

Here 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:

```python
>>> from dask.diagnostics import CacheProfiler
>>> from cachey import nbytes

>>> cprof = CacheProfiler(metric=nbytes)
```

### 4.18.5 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:

```python
>>> 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:
```python
>>> 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:

```python
>>> 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:

```python
>>> 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 (GIL), 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, it shows that these run both concurrently and in parallel, as the CPU percentage spikes up to around 350%.

### 4.18.6 Custom 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

Custom diagnostics can be created either by instantiating the `Callback` class with the some of the above methods as keywords or by subclassing the `Callback` class. Here we create a class that prints the name of every key as it’s computed:

```python
from dask.callbacks import Callback
class PrintKeys(Callback):
    def _pretask(self, key, dask, state):
        print(f'Computing: {repr(key)!r}!')
```

This can now be used as a context manager during computation:

```python
>>> 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`:

```python
>>> def printkeys(key, dask, state):
...     print(f'Computing: {repr(key)!r}!')

>>> with Callback(pretask=printkeys):
...     get(dsk, 'c')
    Computing 'a'!
    Computing 'b'!
    Computing 'c'!
```

### 4.18.7 API
CacheProfiler([metric, metric_name])  A profiler for dask execution at the scheduler cache level.

Callback([start, start_state, pretask, ...])  Base class for using the callback mechanism

Profiler()  A profiler for dask execution at the task level.

ProgressBar([minimum, width, dt, out])  A progress bar for dask.

ResourceProfiler([dt])  A profiler for resource use.

visualize(profilers[, file_path, show, save])  Visualize the results of profiling in a bokeh plot.

dask.diagnostics.ProgressBar (minimum=0, width=40, dt=0.1, out=None)

A progress bar for dask.

Parameters

minimum  [int, optional] Minimum time threshold in seconds before displaying a progress bar. Default is 0 (always display)

width  [int, optional] Width of the bar

dt  [float, optional] Update resolution in seconds, default is 0.1 seconds

Examples

Below we create a progress bar with a minimum threshold of 1 second before displaying. For cheap computations nothing is shown:

```python
>>> with ProgressBar(minimum=1.0):   # doctest: +SKIP
...   out = some_fast_computation.compute()
```

But for expensive computations a full progress bar is displayed:

```python
>>> with ProgressBar(minimum=1.0):   # doctest: +SKIP
...   out = some_slow_computation.compute()
```

The duration of the last computation is available as an attribute

```python
>>> pbar = ProgressBar()
>>> with pbar:   # doctest: +SKIP
...   out = some_computation.compute()
```

```python
[########################################] | 100% Completed | 10.4 s
```

```python
>>> pbar.last_duration   # doctest: +SKIP
```

10.4

You can also register a progress bar so that it displays for all computations:

```python
>>> pbar = ProgressBar()
>>> pbar.register()   # doctest: +SKIP
```

```python
#### Slow computation.compute()
```

```python
[########################################] | 100% Completed | 10.4 s
```

```python
>>> some_slow_computation.compute()  # doctest: +SKIP
```

```python
[########################################] | 100% Completed | 10.4 s
```

dask.diagnostics.Profiler()

A profiler for dask execution at the task level.

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

Examples

```python
>>> from operator import add, mul
>>> from dask.threaded import get
>>> dsk = {'x': 1, 'y': (add, 'x', 10), 'z': (mul, 'y', 2)}
>>> with Profiler() as prof:
...    get(dsk, 'z')
22
```

```python
>>> prof.results # doctest: +SKIP
[('y', (add, 'x', 10), 1435352238.48039, 1435352238.480655, 140285575100160),
 ('z', (mul, 'y', 2), 1435352238.480657, 1435352238.480803, 140285566707456)]
```

These results can be visualized in a bokeh plot using the `visualize` method. Note that this requires bokeh to be installed.

```python
>>> prof.visualize() # doctest: +SKIP
```

You can activate the profiler globally

```python
>>> prof.register() # doctest: +SKIP
```

If you use the profiler globally you will need to clear out old results manually.

```python
>>> prof.clear()
```

`dask.diagnostics.ResourceProfiler(dt=1)`

A profiler for resource use.

**Records the following each timestep**

1. Time in seconds since the epoch
2. Memory usage in MB
3. % CPU usage

Examples

```python
>>> from operator import add, mul
>>> from dask.threaded import get
>>> dsk = {'x': 1, 'y': (add, 'x', 10), 'z': (mul, 'y', 2)}
>>> with ResourceProfiler() as prof: # doctest: +SKIP
...    get(dsk, 'z')
22
```

These results can be visualized in a bokeh plot using the `visualize` method. Note that this requires bokeh to be installed.

```python
>>> prof.visualize() # doctest: +SKIP
```
You can activate the profiler globally

```python
>>> prof.register()  # doctest: +SKIP
```

If you use the profiler globally you will need to clear out old results manually.

```python
>>> prof.clear()  # doctest: +SKIP
```

Note that when used as a context manager data will be collected throughout the duration of the enclosed block. In contrast, when registered globally data will only be collected while a dask scheduler is active.

dask.diagnostics.CacheProfiler(metric=None, metric_name=None)

A profiler for dask execution at the scheduler cache level.

**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

**Examples**

```python
>>> from operator import add, mul
>>> from dask.threaded import get

>>> dsk = {'x': 1, 'y': (add, 'x', 10), 'z': (mul, 'y', 2)}

>>> with CacheProfiler() as prof:
...     get(dsk, 'z')
22

>>> prof.results  # doctest: +SKIP
[CacheData('y', (add, 'x', 10), 1, 1435352238.48039, 1435352238.480655),
 CacheData('z', (mul, 'y', 2), 1, 1435352238.480657, 1435352238.480803)]
```

The default is to count each task (metric is 1 for all tasks). Other functions may 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 cache.

```python
>>> from cachey import nbytes  # doctest: +SKIP

>>> with CacheProfiler(metric=nbytes) as prof:  # doctest: +SKIP
...     get(dsk, 'z')
```

The profiling results can be visualized in a bokeh plot using the visualize method. Note that this requires bokeh to be installed.

```python
>>> prof.visualize()  # doctest: +SKIP
```

You can activate the profiler globally

```python
>>> prof.register()  # doctest: +SKIP
```

If you use the profiler globally you will need to clear out old results manually.
```
>>> prof.clear()
```

```
dask.diagnostics.Callback(start=None, start_state=None, pretask=None, posttask=None, finish=None)
```

Base class for using the callback mechanism

Create a callback with functions of the following signatures:

```
>>> def start(dsk):
...     pass
>>> def start_state(dsk, state):
...     pass
>>> def pretask(key, dsk, state):
...     pass
>>> def posttask(key, result, dsk, state, worker_id):
...     pass
>>> def finish(dsk, state, failed):
...     pass
```

You may then construct a callback object with any number of them

```
>>> cb = Callback(pretask=pretask, finish=finish)  # doctest: +SKIP
```

And use it either as a context manager over a compute/get call

```
>>> with cb:  # doctest: +SKIP
...     x.compute()  # doctest: +SKIP
```

Or globally with the register method

```
>>> cb.register()  # doctest: +SKIP
>>> cb.unregister()  # doctest: +SKIP
```

Alternatively subclass the Callback class with your own methods.

```
>>> class PrintKeys(Callback):
...     def _pretask(self, key, dask, state):
...         print("Computing: {0}!").format(repr(key)))
```

```
>>> with PrintKeys():  # doctest: +SKIP
...     x.compute()  # doctest: +SKIP
```

```
dask.diagnostics.visualize(profilers, file_path=None, show=True, save=True, **kwargs)
```

Visualize the results of profiling in a bokeh plot.

If multiple profilers are passed in, the plots are stacked vertically.

**Parameters**

- `profilers` [profiler or list] Profiler or list of profilers.
- `file_path` [string, optional] Name of the plot output file.
- `show` [boolean, optional] If True (default), the plot is opened in a browser.
- `save` [boolean, optional] If True (default), the plot is saved to disk.
- `**kwargs` Other keyword arguments, passed to bokeh.figure. These will override all defaults set by visualize.

**Returns**
The completed bokeh plot object.

4.19 Diagnostics (distributed)

The *Dask distributed scheduler* provides feedback in two forms:

1. A progress bar suitable for interactive use in consoles or notebooks
2. An interactive dashboard containing several plots and tables with live information

4.19.1 Progress bar

```
progress(*futures, **kwargs) Track progress of futures
```

The `dask.distributed` progress bar differs from the `ProgressBar` used for *local diagnostics*. The `progress` function takes a Dask object that is executing in the background:

```python
# Single machine progress bar
from dask.diagnostics import ProgressBar

with ProgressBar():
    x.compute()

# Distributed scheduler ProgressBar

from dask.distributed import Client, progress

client = Client()  # use dask.distributed by default

x = x.persist()  # start computation in the background
progress(x)  # watch progress

x.compute()  # convert to final result when done if desired
```

4.19.2 Dashboard

```
Client([address, loop, timeout, ...]) Connect to and drive computation on a distributed Dask cluster
```

If Bokeh is installed then the dashboard will start up automatically whenever the scheduler is created. For local use this happens automatically when you create a client with no arguments:

```
from dask.distributed import Client

client = Client()  # start distributed scheduler locally. Launch dashboard
```

It is typically served at [http://localhost:8787/status](http://localhost:8787/status), but may be served elsewhere if this port is taken. The address of the dashboard will be displayed if you are in a Jupyter Notebook.

There are numerous pages with information about task runtimes, communication, statistical profiling, load balancing, memory use, and much more. For more information we recommend the following video guide:
4.19.3 External Documentation

More in-depth technical documentation about Dask’s distributed scheduler is available at https://distributed.dask.org/en/latest

4.19.4 API

dask.distributed.progress(*futures, **kwargs)
   Track progress of futures

   This operates differently in the notebook and the console
   • Notebook: This returns immediately, leaving an IPython widget on screen
   • Console: This blocks until the computation completes

   Parameters

   futures: Futures  A list of futures or keys to track
   notebook: bool (optional)  Running in the notebook or not (defaults to guess)
   multi: bool (optional)  Track different functions independently (defaults to True)
   complete: bool (optional)  Track all keys (True) or only keys that have not yet run (False)
      (defaults to True)

   Notes

   In the notebook, the output of progress must be the last statement in the cell. Typically, this means calling progress at the end of a cell.

   Examples

   >>> progress(futures)  # doctest: +SKIP
   [########################################] | 100% Completed | 1.7s

4.20 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, 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.

4.20.1 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 executes in a separate thread or a different machine, these approaches break down. To address this, 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, so 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.

The single-threaded scheduler can be used, for example, by setting `scheduler='single-threaded'` in a compute call:

```python
>>> x.compute(scheduler='single-threaded')
```

For more ways to configure schedulers, see the [scheduler configuration documentation](https://docs.dask.org/en/latest/configuration.html).

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](https://docs.dask.org/en/latest/testing.html), 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:

```python
>>> x.compute(rerun_exceptions_locally=True)
```

On the distributed scheduler use the `recreate_error_locally` method on anything that contains `Futures`:

```python
>>> 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:

```python
>>> 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.

4.20.2 Inspect Scheduling State

Not all errors present themselves as exceptions. For example, in a distributed system workers may die unexpectedly, 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, workers, and client all emit 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:

```yaml
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.

### 4.20.3 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()`:

```python
>>> 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):

```python
>>> 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:

```python
>>> 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.

### 4.20.4 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.

### Graph Internals

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, \textit{dask.delayed} is often a better choice. If you are a \textit{core developer}, then you should start here.

- \textit{Overview}
- \textit{Specification}
- \textit{Custom Graphs}
- \textit{Optimization}
- \textit{Custom Collections}
- \textit{High Level Graphs}

### 4.21 Overview

An explanation of Dask task graphs.

#### 4.21.1 Motivation

Normally, humans write programs and then compilers/interpreters interpret them (for example, \texttt{python}, \texttt{javac}, \texttt{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 \textit{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 \textit{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.
4.21.2 Example

Consider the following simple program:

```python
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:

```python
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.

4.21.3 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.

4.22 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.

4.22.1 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`.  

---

**4.22. Specification** **815**
4.22.2 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.

4.22.3 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 to a task, as is the case with 'z'. In the latter case, `get` should perform all necessary computation to retrieve the computed value.

```python
>>> 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:

```python
>>> get(dsk, ['x', 'y', 'z'])
[1, 2, 3]
```

Because we accept lists of keys as keys, we support nested lists:

```python
>>> 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.

4.22.4 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:

```python
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:

```python
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.

### 4.23 Custom Graphs

There may be times when 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 to have 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.

#### 4.23.1 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:

```python
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
```

#### 4.23.2 Related Projects

The following excellent projects also provide parallel execution:

- Joblib
- Multiprocessing
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, multiprocessing, 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.

### 4.24 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.

#### 4.24.1 Example

Suppose you had a custom Dask graph for doing a word counting task:

```python
>>> 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)
```

(continues on next page)
Here we are 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, and then returning the output string.

To perform the computation, we first remove unnecessary components from the graph using the `cull` function and then pass the Dask graph and the desired output keys to a scheduler `get` function:

```python
>>> from dask.threaded import get
>>> from dask.optimization import cull

>>> outputs = ['print1', 'print2']
>>> dsk2, _ = cull(dsk, outputs)  # remove unnecessary tasks from the graph
```
As can be seen above, the scheduler computed only the requested outputs ('print3' was never computed). This is because we called the `dask.optimization.cull` function, which removes the unnecessary tasks from the graph.

Culling is part of the default optimization pass of almost all collections. Often you want to call it somewhat early 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)
```
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:

```python
>>> 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:

```python
>>> 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 \texttt{cull}
2. Inlined constants using \texttt{inline}
3. Inlined cheap computations using \texttt{inline\_functions}, improving parallelism
4. Fused linear tasks together to ensure they run on the same worker using \texttt{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 \texttt{dask.optimization}. For more information, see the API below.

### 4.24.2 Rewrite Rules

For context based optimizations, \texttt{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. \texttt{RewriteRule(lhs, rhs, vars)}

   Given a left-hand-side (\texttt{lhs}), a right-hand-side (\texttt{rhs}), and a set of variables (\texttt{vars}), a rewrite rule
   declaratively encodes the following operation:

   \texttt{lhs} \rightarrow \texttt{rhs} if task matches \texttt{lhs} over \texttt{vars}

2. \texttt{RuleSet(*rules)}

   A collection of rewrite rules. The design of \texttt{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:

```python
>>> 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:

```python
# 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.

### 4.24.3 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:

```python
def fuse(dsk, keys=None):
    ...

x.compute(fuse_keys=['x', 'y', 'z'])
```

### 4.24.4 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:
```python
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:

```python
with dask.config.set(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:

```python
with dask.config.set(array_optimize=my_optimize_function,
                     dataframe_optimize=None,
                     delayed_optimize=my_other_optimize_function):
    ...
```

You do not need to specify all collections. Collections will default to their standard optimization scheme (which is usually a good choice).

### 4.24.5 API

#### Top level optimizations

- `cull(dsk, keys)`
  Return new dask with only the tasks required to calculate keys. In other words, remove unnecessary tasks from dask. `keys` may be a single key or list of keys.

  Returns
  
  `dsk`: culled dask graph

- `fuse(dsk[, keys, dependencies, ave_width, ...])`
  Fuse tasks that form reductions; more advanced than `fuse_linear`

- `inline(dsk[, keys, inline_constants, ...])`
  Return new dask with the given keys inlined with their values.

- `inline_functions(dsk, output[, ...])`
  Inline cheap functions into larger operations

#### Utility functions

- `functions_of(task)`
  Set of functions contained within nested task

#### Rewrite Rules

- `RewriteRule(lhs, rhs[, vars])`
  A rewrite rule.

- `RuleSet(*rules)`
  A set of rewrite rules.

#### Definitions

- `dask.optimization.cull(dsk, keys)`
  Return new dask with only the tasks required to calculate keys.

  In other words, remove unnecessary tasks from dask. `keys` may be a single key or list of keys.

  Returns

  `dsk`: culled dask graph
dependencies: Dict mapping {key: [deps]}. Useful side effect to accelerate other optimizations, notably fuse.

Examples

```python
>>> d = {'x': 1, 'y': (inc, 'x'), 'out': (add, 'x', 10)}
>>> dsk, dependencies = cull(d, 'out')  # doctest: +SKIP
>>> dsk  # doctest: +SKIP
{'x': 1, 'out': (add, 'x', 10)}
>>> dependencies  # doctest: +SKIP
{'x': set(), 'out': set(['x'])}
```

dask.optimization.fuse(dsk, keys=None, dependencies=None, ave_width=None, max_width=None, max_height=None, max_depth_new_edges=None, rename_keys=None, fuse_subgraphs=None)

Fuse tasks that form reductions; more advanced than fuse_linear

This trades parallelism opportunities for faster scheduling by making tasks less granular. It can replace fuse_linear in optimization passes.

This optimization applies to all reductions–tasks that have at most one dependent–so it may be viewed as fusing “multiple input, single output” groups of tasks into a single task. There are many parameters to fine tune the behavior, which are described below. ave_width is the natural parameter with which to compare parallelism to granularity, so it should always be specified. Reasonable values for other parameters with be determined using ave_width if necessary.

Parameters

dsk: dict  dask graph

keys: list or set, optional  Keys that must remain in the returned dask graph

dependencies: dict, optional  {key: [list-of-keys]}. Must be a list to provide count of each key This optional input often comes from cull

ave_width: float (default 2)  Upper limit for width = num_nodes / height, a good measure of parallelizability

max_width: int  Don’t fuse if total width is greater than this

max_height: int  Don’t fuse more than this many levels

max_depth_new_edges: int  Don’t fuse if new dependencies are added after this many levels

rename_keys: bool or func, optional  Whether to rename the fused keys with default_fused_keys_renamer or not. Renaming fused keys can keep the graph more understandable and comprehensive, but it comes at the cost of additional processing. If False, then the top-most key will be used. For advanced usage, a function to create the new name is also accepted.

fuse_subgraphs  [bool, optional] Whether to fuse multiple tasks into SubgraphCallable objects.

Returns

dsk: output graph with keys fused

dependencies: dict mapping dependencies after fusion. Useful side effect to accelerate other downstream optimizations.
dask.optimization.inline(dsk, keys=None, inline_constants=True, dependencies=None)

Return new dask with the given keys inlined with their values.

Inlines all constants if inline_constants keyword is True. Note that the constant keys will remain in the graph, to remove them follow inline with cull.

Examples

```python
>>> d = {'x': 1, 'y': (inc, 'x'), 'z': (add, 'x', 'y')}
```

```python
>>> inline(d)  # doctest: +SKIP
{'x': 1, 'y': (inc, 1), 'z': (add, 1, 'y')}
```

```python
>>> inline(d, keys='y')  # doctest: +SKIP
{'x': 1, 'y': (inc, 1), 'z': (add, 1, (inc, 1))}
```

```python
>>> inline(d, keys='y', inline_constants=False)  # doctest: +SKIP
{'x': 1, 'y': (inc, 1), 'z': (add, 'x', (inc, 'x'))}
```

dask.optimization.inline_functions(dsk, output, fast_functions=None, inline_constants=False, dependencies=None)

Inline cheap functions into larger operations

Examples

```python
>>> dsk = {'out': (add, 'i', 'd'),  # doctest: +SKIP
...        'i': (inc, 'x'),
...        'd': (double, 'y'),
...        'x': 1, 'y': 1}
```

```python
>>> inline_functions(dsk, [], [inc])  # doctest: +SKIP
{'out': (add, (inc, 'x'), 'd'),
 'd': (double, 'y'),
 'x': 1, 'y': 1}
```

Protect output keys. In the example below i is not inlined because it is marked as an output key.

```python
>>> inline_functions(dsk, ['i', 'out'], [inc, double])  # doctest: +SKIP
{'out': (add, (inc, 'x'), 'd'),
 'i': (inc, 'x'),
 'x': 1, 'y': 1}
```

dask.optimization.functions_of(task)

Set of functions contained within nested task

Examples

```python
>>> task = (add, (mul, 1, 2), (inc, 3))  # doctest: +SKIP
```

```python
>>> functions_of(task)  # doctest: +SKIP
set([add, mul, inc])
```

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.

```python
>>> 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.

```python
>>> 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

```python
>>> def f(*args): pass
>>> def g(*args): pass
>>> def h(*args): pass
>>> from operator import add

>>> rs = RuleSet(
...     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
```
```python
>>> rs.rewrite((f, (g, 'a', 3)))  # doctest: +SKIP
(h, 'a', 3)
```

```python
>>> dsk = {'a': (add, 2, 0),  # Apply ruleset to full dask graph
...         'b': (f, (g, 'a', 3))}
```

```python
>>> from toolz import valmap
>>> valmap(rs.rewrite, dsk)  # doctest: +SKIP
{'a': 2,  
'b': (h, 'a', 3)}
```

**Attributes**

- `rules` [list] A list of `RewriteRule`s included in the `RuleSet`.

### 4.25 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*

#### 4.25.1 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 is 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.

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.

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 instance method.

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.

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.


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.:

```python
>>> import dask.threaded
>>> class MyCollection(object):
...     # Use the threaded scheduler by default
...     __daskScheduler__ = 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.

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`.

**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*.

---

### 4.25.2 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.

---

### 4.25. Custom Collections

---

831
Once the appropriate scheduler `get` function is determined, it is 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 the following:

```python
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]
            # `kwars` 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 `kwars` 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 rebuilder for each collection is called on the graph for its respective results.

In pseudocode, this looks like the following:

```python
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)
```

4.25. Custom Collections 833
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 the following:

```python
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)

    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 outputs to the specified file.

In pseudocode, this looks like the following:

```python
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)
```
4.25.3 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, you can subclass from `dask.base.DaskMethodsMixin` which adds implementations of `compute`, `persist`, and `visualize` based on the interface above.

4.25.4 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`:

```python
# 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):
```

(continues on next page)
# 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:

```python
>>> 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)
```

### 4.25.5 Checking if an object is a Dask collection

To check if an object is a Dask collection, use `dask.base.is_dask_collection`:

```python
>>> 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
```
4.25.6 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:

1. The `__dask_tokenize__` method

   Where possible, it is 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, where only the location of the definition is different.

Note: Both Dask collections and normal Python objects can have implementations of `tokenize` using either of the methods described above.

Example

```python
>>> 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)
```

(continues on next page)
For more examples, see `dask/base.py` or any of the built-in Dask collections.

### 4.26 High Level Graphs

Dask graphs produced by collections like Arrays, Bags, and DataFrames have high-level structure that can be useful for visualization and high-level optimization. The task graphs produced by these collections encode this structure explicitly as `HighLevelGraph` objects. This document describes how to work with these in more detail.

#### 4.26.1 Motivation and Example

In full generality, Dask schedulers expect arbitrary task graphs where each node is a single Python function call and each edge is a dependency between two function calls. These are usually stored in flat dictionaries. Here is some simple Dask DataFrame code and the task graph that it might generate:

```python
import dask.dataframe as dd

df = dd.read_csv('myfile.*.csv')
df = df + 100
df = df[df.name == 'Alice']
```

```json
{
    ('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
    ('read-csv', 1): (pandas.read_csv, 'myfile.1.csv'),
    ('read-csv', 2): (pandas.read_csv, 'myfile.2.csv'),
    ('read-csv', 3): (pandas.read_csv, 'myfile.3.csv'),
    ('add', 0): (operator.add, 'myfile.0.csv', 100),
    ('add', 1): (operator.add, 'myfile.1.csv', 100),
    ('add', 2): (operator.add, 'myfile.2.csv', 100),
    ('add', 3): (operator.add, 'myfile.3.csv', 100),
    ('filter', 0): (lambda part: part[part.name == 'Alice'], ('add', 0)),
    ('filter', 1): (lambda part: part[part.name == 'Alice'], ('add', 1)),
    ('filter', 2): (lambda part: part[part.name == 'Alice'], ('add', 2)),
    ('filter', 3): (lambda part: part[part.name == 'Alice'], ('add', 3)),
}
```

The task graph is a dictionary that stores every Pandas-level function call necessary to compute the final result. We can see that there is some structure to this dictionary if we separate out the tasks that were associated to each high-level Dask DataFrame operation:

```json
{
    # From the dask.dataframe.read_csv call
    ('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
    ('read-csv', 1): (pandas.read_csv, 'myfile.1.csv'),
    ('read-csv', 2): (pandas.read_csv, 'myfile.2.csv'),
    ('read-csv', 3): (pandas.read_csv, 'myfile.3.csv'),
}
(continues on next page)
By understanding this high-level structure we are able to understand our task graphs more easily (this is more important for larger datasets when there are thousands of tasks per layer) and how to perform high-level optimizations. For example, in the case above we may want to automatically rewrite our code to filter our datasets before adding 100:

# Before
```
df = dd.read_csv('myfile.*.csv')
df = df + 100
df = df[df.name == 'Alice']
```

# After
```
df = dd.read_csv('myfile.*.csv')
df = df[df.name == 'Alice']
df = df + 100
```

Dask’s high level graphs help us to explicitly encode this structure by storing our task graphs in layers with dependencies between layers:

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

>>> df = dd.read_csv('myfile.*.csv')
>>> df = df + 100
>>> df = df[df.name == 'Alice']

>>> graph = df.__dask_graph__()

>>> graph.layers

| 'read-csv': {
  ('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
  ('read-csv', 1): (pandas.read_csv, 'myfile.1.csv'),
  ('read-csv', 2): (pandas.read_csv, 'myfile.2.csv'),
  ('read-csv', 3): (pandas.read_csv, 'myfile.3.csv')
}

| 'add': {
  ('add', 0): (operator.add, 'myfile.0.csv', 100),
  ('add', 1): (operator.add, 'myfile.1.csv', 100),
  ('add', 2): (operator.add, 'myfile.2.csv', 100),
  ('add', 3): (operator.add, 'myfile.3.csv', 100)
}

| 'filter': {
  ('filter', 0): (lambda part: part[part.name == 'Alice'], ('add', 0)),
  ('filter', 1): (lambda part: part[part.name == 'Alice'], ('add', 1)),
  ('filter', 2): (lambda part: part[part.name == 'Alice'], ('add', 2)),
  ('filter', 3): (lambda part: part[part.name == 'Alice'], ('add', 3))
}
```

```
While the DataFrame points to the output layers on which it depends directly:

```python
>>> df.__dask_layers__()
{'filter'}
```

4.26.2 HighLevelGraphs

The `HighLevelGraph` object is a `Mapping` object composed of other sub-Mappings, along with a high-level dependency mapping between them:

```python
class HighLevelGraph(Mapping):
    layers: Dict[str, Mapping]
    dependencies: Dict[str, Set[str]]
```

You can construct a HighLevelGraph explicitly by providing both to the constructor:

```python
layers = {
    'read-csv': {('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
                  ('read-csv', 1): (pandas.read_csv, 'myfile.1.csv'),
                  ('read-csv', 2): (pandas.read_csv, 'myfile.2.csv'),
                  ('read-csv', 3): (pandas.read_csv, 'myfile.3.csv')},
    'add': {('add', 0): (operator.add, 'myfile.0.csv', 100),
             ('add', 1): (operator.add, 'myfile.1.csv', 100),
             ('add', 2): (operator.add, 'myfile.2.csv', 100),
             ('add', 3): (operator.add, 'myfile.3.csv', 100)},
    'filter': {('filter', 0): (lambda part: part[part.name == 'Alice'], ('add', 0)),
                ('filter', 1): (lambda part: part[part.name == 'Alice'], ('add', 1)),
                ('filter', 2): (lambda part: part[part.name == 'Alice'], ('add', 2)),
                ('filter', 3): (lambda part: part[part.name == 'Alice'], ('add', 3))}
}
dependencies = {'read-csv': set(),
                 'add': {'read-csv'},
                 'filter': {'add'}}

graph = HighLevelGraph(layers, dependencies)
```

This object satisfies the `Mapping` interface, and so operates as a normal Python dictionary that is the semantic merger of the underlying layers:

```python
>>> len(graph)
12
>>> graph[('read-csv', 0)]
('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
```
4.26.3 API

class dask.highlevelgraph.HighLevelGraph(layers, dependencies)
Task graph composed of layers of dependent subgraphs

This object encodes a Dask task graph that is composed of layers of dependent subgraphs, such as commonly occurs when building task graphs using high level collections like Dask array, bag, or dataframe.

Typically each high level array, bag, or dataframe operation takes the task graphs of the input collections, merges them, and then adds one or more new layers of tasks for the new operation. These layers typically have at least as many tasks as there are partitions or chunks in the collection. The HighLevelGraph object stores the subgraphs for each operation separately in sub-graphs, and also stores the dependency structure between them.

Parameters

layers [Dict[str, Mapping]] The subgraph layers, keyed by a unique name

dependencies [Dict[str, Set[str]]] The set of layers on which each layer depends

See also:

HighLevelGraph.from_collections typically used by developers to make new HighLevelGraphs

Examples

Here is an idealized example that shows the internal state of a HighLevelGraph

```python
>>> import dask.dataframe as dd

>>> df = dd.read_csv('myfile.*.csv')  # doctest: +SKIP
>>> df = df + 100  # doctest: +SKIP
>>> df = df[df.name == 'Alice']  # doctest: +SKIP

>>> graph = df.__dask_graph__()  # doctest: +SKIP

>>> graph.layers  # doctest: +SKIP
{'read-csv': {('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
             ('read-csv', 1): (pandas.read_csv, 'myfile.1.csv'),
             ('read-csv', 2): (pandas.read_csv, 'myfile.2.csv'),
             ('read-csv', 3): (pandas.read_csv, 'myfile.3.csv')},
 'add': {('add', 0): (operator.add, ('read-csv', 0), 100),
         ('add', 1): (operator.add, ('read-csv', 1), 100),
         ('add', 2): (operator.add, ('read-csv', 2), 100),
         ('add', 3): (operator.add, ('read-csv', 3), 100)},
 'filter': {('filter', 0): (lambda part: part[part.name == 'Alice'], ('add', 0)),
            ('filter', 1): (lambda part: part[part.name == 'Alice'], ('add', 1)),
            ('filter', 2): (lambda part: part[part.name == 'Alice'], ('add', 2)),
            ('filter', 3): (lambda part: part[part.name == 'Alice'], ('add', 3))}}

>>> graph.dependencies  # doctest: +SKIP
{'read-csv': set(),
 'add': {'read-csv'},
 'filter': {'add'}}
```
classmethod from_collections(name, layer, dependencies=())
Construct a HighLevelGraph from a new layer and a set of collections

This constructs a HighLevelGraph in the common case where we have a single new layer and a set of old
collections on which we want to depend.

This pulls out the __dask_layers__() method of the collections if they exist, and adds them to the
dependencies for this new layer. It also merges all of the layers from all of the dependent collections
together into the new layers for this graph.

Parameters

name [str] The name of the new layer
layer [Mapping] The graph layer itself

Examples

In typical usage we make a new task layer, and then pass that layer along with all dependent collections
to this method.

```python
>>> def add(self, other):
...     name = 'add-' + tokenize(self, other)
...     layer = {(name, i): (add, input_key, other)
...             for i, input_key in enumerate(self.__dask_keys__())}
...     graph = HighLevelGraph.from_collections(name, layer,
...                                           dependencies=[self])
...     return new_collection(name, graph)
```

get(k[, d]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D’s items

keys() → a set-like object providing a view on D’s keys

values() → an object providing a view on D’s values

Help & reference

- Development Guidelines
- Changelog
- Configuration
- Presentations On Dask
- Dask Cheat Sheet
- Comparison to Spark
- Opportunistic Caching
- Internal Data Ingestion
- Remote Data
- GPUs
- Citations
- Funding
4.27 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.

4.27.1 Where to ask for help

Dask conversation happens in the following places:

1. Stack Overflow #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 Stack Overflow and GitHub issues over gitter chat. GitHub and Stack Overflow are more easily searchable by future users and so is more efficient for everyone’s time. Gitter chat is generally reserved for community discussion.

4.27.2 Separate Code Repositories

Dask maintains code and documentation in a few git repositories hosted on the GitHub dask organization, https://github.com/dask. This includes the primary repository and several other repositories for different components. A non-exhaustive list follows:

- https://github.com/dask/dask: The main code repository holding parallel algorithms, the single-machine scheduler, and most documentation
- https://github.com/dask/distributed: The distributed memory scheduler
- https://github.com/dask/s3fs: S3 Filesystem interface
- https://github.com/dask/gcsfs: GCS Filesystem interface
- https://github.com/dask/hdfs3: Hadoop Filesystem interface
- ...

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 https://pandas.pydata.org/pandas-docs/stable/contributing.html

4.27.3 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 “good first issue” label, which contains issues that are good for starting developers. Generally, familiarity with Python, NumPy, Pandas, and some parallel computing are assumed.
4.27.4 Development Environment

**Download code**

Make a fork of the main Dask repository and clone the fork:

```
$ git clone https://github.com/<your-github-username>/dask
```

Contributions to Dask can then be made by submitting pull requests on GitHub.

**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
```

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 --doctest-modules
```

4.27.5 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 3.5, 3.6, and 3.7. 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 and Appveyor continuous testing frameworks on every push to every pull request on GitHub.

Tests are organized within the various modules’ subdirectories:

```
dask/array/tests/test_*.py
dask/bag/tests/test_*.py
dask/dataframe/tests/test_*.py
dask/diagnostics/tests/test_*.py
```

For the Dask collections like Dask Array and Dask DataFrame, behavior is typically tested directly against the NumPy or Pandas libraries using the `assert_eq` functions:

```
import numpy as np
import dask.array as da
from dask.array.utils import assert_eq

def test_aggregations():
    nx = np.random.random(100)
    dx = da.from_array(x, chunks=(10,))

    assert_eq(nx.sum(), dx.sum())
    assert_eq(nx.min(), dx.min())
    assert_eq(nx.max(), dx.max())
    ...
```

This technique helps to ensure compatibility with upstream libraries and tends to be simpler than testing correctness directly. Additionally, by passing Dask collections directly to the `assert_eq` function rather than call compute manually, the testing suite is able to run a number of checks on the lazy collections themselves.

**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.

```python
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 3.6 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
```

### 4.27.6 Contributing to Documentation

Dask uses Sphinx for documentation, hosted on https://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 the necessary requirements:

```
cd docs/
pip install -r requirements-docs.txt
```

Then build the documentation with `make`:
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.

4.28 Changelog

4.28.1 1.2.2 / 2019-05-08

Array

• Clarify regions kwarg to array.store (GH#4759) Martin Durant
• Add dtype= parameter to da.random.randint (GH#4753) Matthew Rocklin
• Use “row major” rather than “C order” in docstring (GH#4452) @asmith26
• Normalize Xarray datasets to Dask arrays (GH#4756) Matthew Rocklin
• Remove normed keyword in da.histogram (GH#4755) Matthew Rocklin

Bag

• Add key argument to Bag.distinct (GH#4423) Daniel Severo

Core

• Add core dask config file (GH#4774) Matthew Rocklin
• Add core dask config file to MANIFEST.in (GH#4780) James Bourbeau
• Enabling glob with HTTP file-system (GH#3926) Martin Durant
• HTTPFile.seek with whence=1 (GH#4751) Martin Durant
• Remove config key normalization (GH#4742) Jim Crist

Dataframe

• Remove explicit references to Pandas in dask.dataframe.groupby (GH#4778) Matthew Rocklin
• Add support for group_keys kwarg in DataFrame.groupby() (GH#4771) Brian Chu
• Describe doc (GH#4762) Martin Durant
• Remove explicit pandas check in cumulative aggregations (GH#4765) Nick Becker
• Added meta for read_json and test (GH#4588) Abhinav Ralhan
• Add test for dtype casting (GH#4760) Martin Durant
• Document alignment in map_partitions (GH#4757) Jim Crist
• Implement Series.str.split(expand=True) (GH#4744) Matthew Rocklin
Documentation

- Tweaks to develop.rst from trying to run tests (GH#4772) Christian Hudon
- Add document describing phases of computation (GH#4766) Matthew Rocklin
- Point users to Dask-Yarn from spark documentation (GH#4770) Matthew Rocklin
- Update images in delayed doc to remove labels (GH#4768) Martin Durant
- Explain intermediate storage for dask arrays (GH#4025) John A Kirkham
- Specify bash code-block in array best practices (GH#4764) James Bourbeau
- Add array best practices doc (GH#4705) Matthew Rocklin
- Update optimization docs now that cull is not automatic (GH#4752) Matthew Rocklin

4.28.2 1.2.1 / 2019-04-29

Array

- Fix map_blocks with block_info and broadcasting (GH#4737) Bruce Merry
- Make ‘minlength’ keyword argument optional in da.bincount (GH#4684) Genevieve Buckley
- Add support for map_blocks with no array arguments (GH#4713) Bruce Merry
- Add dask.array.trace (GH#4717) Danilo Horta
- Add sizeof support for cupy.ndarray (GH#4715) Peter Andreas Entschev
- Add name kwarg to from_zarr (GH#4663) Michael Eaton
- Add chunks=’auto’ to from_array (GH#4704) Matthew Rocklin
- Raise TypeError if dask array is given as shape for da.ones, zeros, empty or full (GH#4707) Genevieve Buckley
- Add TileDB backend (GH#4679) Isaiah Norton

Core

- Delay long list arguments (GH#4735) Matthew Rocklin
- Bump to numpy >= 1.13, pandas >= 0.21.0 (GH#4720) Jim Crist
- Remove file “test” (GH#4710) James Bourbeau
- Reenable development build, uses upstream libraries (GH#4696) Peter Andreas Entschev
- Remove assertion in HighLevelGraph constructor (GH#4699) Matthew Rocklin

DataFrame

- Change cum-aggregation last-nonnull-value algorithm (GH#4736) Nick Becker
- Fixup series-groupby-apply (GH#4738) Jim Crist
- Refactor array.percentile and dataframe.quantile to use t-digest (GH#4677) Janne Vuorela
- Allow naive concatenation of sorted dataframes (GH#4725) Matthew Rocklin
- Fix perf issue in dd.Series.isin (GH#4727) Jim Crist
- Remove hard pandas dependency for melt by using methodcaller (GH#4719) Nick Becker
- A few dataframe metadata fixes (GH#4695) Jim Crist
- Add DataFrame.replace (GH#4714) Matthew Rocklin
- Add ‘threshold’ parameter to pd.DataFrame.dropna (GH#4625) Nathan Matare

**Documentation**

- Add warning about derived docstrings early in the docstring (GH#4716) Matthew Rocklin
- Create dataframe best practices doc (GH#4703) Matthew Rocklin
- Uncomment dask_sphinx_theme (GH#4728) James Bourbeau
- Fix minor typo fix in a Queue/fire_and_forget example (GH#4709) Matthew Rocklin
- Update from_pandas docstring to match signature (GH#4698) James Bourbeau

**4.28.3 1.2.0 / 2019-04-12**

**Array**

- Fixed mean() and moment() on sparse arrays (GH#4525) Peter Andreas Entschev
- Add test for NEP-18. (GH#4675) Hameer Abbasi
- Allow None to say “no chunking” in normalize_chunks (GH#4656) Matthew Rocklin
- Fix limit value in auto_chunks (GH#4645) Matthew Rocklin

**Core**

- Updated diagnostic bokeh test for compatibility with bokeh>=1.1.0 (GH#4680) Philipp Rudiger
- Adjusts codecov’s target/threshold, disable patch (GH#4671) Peter Andreas Entschev
- Always start with empty http buffer, not None (GH#4673) Martin Durant

**DataFrame**

- Propagate index dtype and name when create dask dataframe from array (GH#4686) Henrique Ribeiro
- Fix ordering of quantiles in describe (GH#4647) gregrf
- Clean up and document rearrange_column_by_tasks (GH#4674) Matthew Rocklin
- Mark some parquet tests xfail (GH#4667) Peter Andreas Entschev
- Fix parquet breakages with arrow 0.13.0 (GH#4668) Martin Durant
- Allow sample to be False when reading CSV from a remote URL (GH#4634) Ian Rose
- Fix timezone metadata inference on parquet load (GH#4655) Martin Durant
- Use is_dataframe/index_like in dd.utils (GH#4657) Matthew Rocklin
- Add min_count parameter to groupby sum method (GH#4648) Henrique Ribeiro
- Correct quantile to handle unsorted quantiles (GH#4650) gregrf
Dask Documentation, Release 1.2.2

Documentation

- Add delayed extra dependencies to install docs (GH#4660) James Bourbeau

4.28.4 1.1.5 / 2019-03-29

Array

- Ensure that we use the dtype keyword in normalize_chunks (GH#4646) Matthew Rocklin

Core

- Use recursive glob in LocalFileSystem (GH#4186) Brett Naul
- Avoid YAML deprecation (GH#4603)
- Fix CI and add set -e (GH#4605) James Bourbeau
- Support builtin sequence types in dask.visualize (GH#4602)
- unpack/repack orderedDict (GH#4623) Justin Poehnelt
- Add da.random.randint to API docs (GH#4628) James Bourbeau
- Add zarr to CI environment (GH#4604) James Bourbeau
- Enable codecov (GH#4631) Peter Andreas Entschev

DataFrame

- Support setting the index (GH#4565)
- DataFrame.itertuples accepts index, name kwargs (GH#4593) Dan O’Donovan
- Support non-Pandas series in dd.Series.unique (GH#4599) Ben Zaitlen
- Replace use of explicit type check with _is_partition_type predicate (GH#4533)
- Remove additional pandas warnings in tests (GH#4576)
- Check object for name/dtype attributes rather than type (GH#4606)
- Fix comparison against pd.Series (GH#4613) amerkel2
- Fixing warning from setting categorical codes to floats (GH#4624) Julia Signell
- Fix renaming on index to_frame method (GH#4498) Henrique Ribeiro
- Fix divisions when joining two single-partition dataframes (GH#4636) Justin Waugh
- Warn if partitions overlap in compute_divisions (GH#4600) Brian Chu
- Give informative meta= warning (GH#4637) Matthew Rocklin
- Add informative error message to Series._getitem__ (GH#4638) Matthew Rocklin
- Add clear exception message when using index or index_col in read_csv (GH#4651) Álvaro Abella Bascarán
Documentation

- Add documentation for custom groupby aggregations (GH#4571)
- Docs dataframe joins (GH#4569)
- Specify fork-based contributions (GH#4619) James Bourbeau
- correct to_parquet example in docs (GH#4641) Aaron Fowles
- Update and secure several references (GH#4649) Søren Fuglede Jørgensen

4.28.5 1.1.4 / 2019-03-08

Array

- Use mask selection in compress (GH#4548) John A Kirkham
- Use asarray in extract (GH#4549) John A Kirkham
- Use correct dtype when test concatenation. (GH#4539) Elliott Sales de Andrade
- Fix CuPy tests or properly marks as xfail (GH#4564) Peter Andreas Entschev

Core

- Fix local scheduler callback to deal with custom caching (GH#4542) Yu Feng
- Use parse_bytes in read_bytes(sample=…) (GH#4554) Matthew Rocklin

DataFrame

- Fix up groupby-standard deviation again on object dtype keys (GH#4541) Matthew Rocklin
- TST/CI: Updates for pandas 0.24.1 (GH#4551) Tom Augspurger
- Add ability to control number of unique elements in timeseries (GH#4557) Matthew Rocklin
- Add support in read_csv for parameter skiprows for other iterables (GH#4560) @JulianWgs

Documentation

- DataFrame to Array conversion and unknown chunks (GH#4516) Scott Sievert
- Add docs for random array creation (GH#4566) Matthew Rocklin
- Fix typo in docstring (GH#4572) Shyam Saladi

4.28.6 1.1.3 / 2019-03-01

Array

- Modify mean chunk functions to return dicts rather than arrays (GH#4513) Matthew Rocklin
- Change sparse installation in CI for NumPy/Python2 compatibility (GH#4537) Matthew Rocklin
DataFrame

- Make merge dispatchable on pandas/other dataframe types (GH#4522) Matthew Rocklin
- `read_sql_table` - datetime index fix and index type checking (GH#4474) Joe Corbett
- Use generalized form of index checking (is_index_like) (GH#4531) Ben Zaitlen
- Add tests for groupby reductions with object dtypes (GH#4535) Matthew Rocklin
- Fixes #4467 : Updates time_series for pandas deprecation (GH#4530) @HSR05

Documentation

- Add missing method to documentation index (GH#4528) Bart Broere

4.28.7 1.1.2 / 2019-02-25

Array

- Fix another unicode/mixed-type edge case in normalize_array (GH#4489) Marco Neumann
- Add `dask.array.diagonal` (GH#4431) Danilo Horta
- Call asanyarray in unify_chunks (GH#4506) Jim Crist
- Modify moment chunk functions to return dicts (GH#4519) Peter Andreas Entschev

Bag

- Don’t inline output keys in `dask.bag` (GH#4464) Jim Crist
- Ensure that `bag.from_sequence` always includes at least one partition (GH#4475) Anderson Banihirwe
- Implement out_type for `bag.fold` (GH#4502) Matthew Rocklin
- Remove map from bag keynames (GH#4500) Matthew Rocklin
- Avoid `itertools.repeat` in `map_partitions` (GH#4507) Matthew Rocklin

DataFrame

- Fix relative path parsing on windows when using fastparquet (GH#4445) Janne Vuorela
- Fix bug in pyarrow and hdfs (GH#4453) (GH#4455) Michal Jastrzębski
- `df getitem` with integer slices is not implemented (GH#4466) Jim Crist
- Replace cudf-specific code with dask-cudf import (GH#4470) Matthew Rocklin
- Avoid groupby.agg(callable) in groupby-var (GH#4482) Matthew Rocklin
- Consider uint types as numerical in check_meta (GH#4485) Marco Neumann
- Fix some typos in groupby comments (GH#4494) Daniel Saxton
- Add error message around `set_index(inplace=True)` (GH#4501) Matthew Rocklin
- `meta_nonempty` works with categorical index (GH#4505) Jim Crist
- Add module name to expected meta error message (GH#4499) Matthew Rocklin
• groupby-nunique works on empty chunk (GH#4504) Jim Crist
• Propogate index metadata if not specified (GH#4509) Jim Crist

Documentation

• Update docs to use from_zarr (GH#4472) John A Kirkham
• DOC: add section of Using Other S3-Compatible Services for remote-data-services (GH#4405) Aploium
• Fix header level of section in changelog (GH#4483) Bruce Merry
• Add quotes to pip install [skip-ci] (GH#4508) James Bourbeau

Core

• Extend started_cbs AFTER state is initialized (GH#4460) Marco Neumann
• Fix bug in HTTPFile._fetch_range with headers (GH#4479) (GH#4480) Ross Petchler
• Repeat optimize_blockwise for diamond fusion (GH#4492) Matthew Rocklin

4.28.8 1.1.1 / 2019-01-31

Array

• Add support for cupy.einsum (GH#4402) Johnnie Gray
• Provide byte size in chunks keyword (GH#4434) Adam Beberg
• Raise more informative error for histogram bins and range (GH#4430) James Bourbeau

DataFrame

• Lazily register more cudf functions and move to backends file (GH#4396) Matthew Rocklin
• Fix ORC tests for pyarrow 0.12.0 (GH#4413) Jim Crist
• rearrange_by_column: ensure that shuffle arg defaults to ‘disk’ if it’s None in dask.config (GH#4414) George Sakkis
• Implement filters for _read_pyarrow (GH#4415) George Sakkis
• Avoid checking against types in is_dataframe_like (GH#4418) Matthew Rocklin
• Pass username as ‘user’ when using pyarrow (GH#4438) Roma Sokolov

Delayed

• Fix DelayedAttr return value (GH#4440) Matthew Rocklin

Documentation

• Use SVG for pipeline graphic (GH#4406) John A Kirkham
• Add doctest-modules to py.test documentation (GH#4427) Daniel Severo
Core

- Work around psutil 5.5.0 not allowing pickling Process objects Janne Vuorela

**4.28.9 1.1.0 / 2019-01-18**

Array

- Fix the average function when there is a masked array (GH#4236) Damien Garaud
- Add allow_unknown_chunksize to hstack and vstack (GH#4287) Paul Vecchio
- Fix tensordot for 27+ dimensions (GH#4304) Johnnie Gray
- Fixed block_info with axes. (GH#4301) Tom Augspurger
- Use safe_wraps for matmul (GH#4346) Mark Harfouche
- Use chunks="auto" in array creation routines (GH#4354) Matthew Rocklin
- Fix np.matmul in dask.array.Array.__array_ufunc__ (GH#4363) Stephan Hoyer
- COMPAT: Re-enable multifield copy->view change (GH#4357) Diane Trout
- Calling np.dtype on a delayed object works (GH#4387) Jim Crist
- Rework normalize_array for numy data (GH#4312) Marco Neumann

DataFrame

- Add fill_value support for series comparisons (GH#4250) James Bourbeau
- Add schema name in read_sql_table for empty tables (GH#4268) Mina Farid
- Adjust check for bad chunks in map_blocks (GH#4308) Tom Augspurger
- Add dask.dataframe.read_fwf (GH#4316) @slnguyen
- Use atop fusion in dask dataframe (GH#4229) Matthew Rocklin
- Use parallel_types() in from_pandas (GH#4331) Matthew Rocklin
- Change DataFrame._repr_data to method (GH#4330) Matthew Rocklin
- Install pyarrow fastparquet for Appveyor (GH#4338) Gábor Lipták
- Remove explicit pandas checks and provide cudf lazy registration (GH#4359) Matthew Rocklin
- Replace isinstance(…, pandas) with is_dataframe_like (GH#4375) Matthew Rocklin
- ENH: Support 3rd-party ExtensionArrays (GH#4379) Tom Augspurger
- Pandas 0.24.0 compat (GH#4374) Tom Augspurger

Documentation

- Fix link to ‘map_blocks’ function in array api docs (GH#4258) David Hoese
- Add a paragraph on Dask-Yarn in the cloud docs (GH#4260) Jim Crist
• Copy edit documentation (GH#4267), (GH#4263), (GH#4262), (GH#4277), (GH#4271), (GH#4279), (GH#4265), (GH#4295), (GH#4293), (GH#4296), (GH#4302), (GH#4306), (GH#4318), (GH#4314), (GH#4309), (GH#4317), (GH#4326), (GH#4325), (GH#4322), (GH#4332), (GH#4333), Miguel Farrajota
• Fix typo in code example (GH#4272) Daniel Li
• Doc: Update array-api.rst (GH#4259) (GH#4282) Prabakaran Kumaresshan
• Update hpc doc (GH#4266) Guillaume Eynard-Bontemps
• Doc: Replace from_avro with read_avro in documents (GH#4313) Prabakaran Kumaresshan
• Remove reference to “get” scheduler functions in docs (GH#4350) Matthew Rocklin
• Fix typo in docstring (GH#4376) Daniel Saxton
• Added documentation for dask.dataframe.merge (GH#4382) Jendrik Jördening

Core

• Avoid recursion in dask.core.get (GH#4219) Matthew Rocklin
• Remove verbose flag from pytest setup.cfg (GH#4281) Matthew Rocklin
• Support Pytest 4.0 by specifying marks explicitly (GH#4280) Takahiro Kojima
• Add High Level Graphs (GH#4092) Matthew Rocklin
• Fix SerializableLock locked and acquire methods (GH#4294) Stephan Hoyer
• Pin boto3 to earlier version in tests to avoid moto conflict (GH#4276) Martin Durant
• Treat None as missing in config when updating (GH#4324) Matthew Rocklin
• Update Appveyor to Python 3.6 (GH#4337) Gábor Lipták
• Use parse_bytes more liberally in dask.dataframe/bytes/bag (GH#4339) Matthew Rocklin
• Add a better error message when cloudpickle is missing (GH#4342) Mark Harfouche
• Support pool= keyword argument in threaded/multiprocessing get functions (GH#4351) Matthew Rocklin
• Allow updates from arbitrary Mappings in config.update, not only dicts. (GH#4356) Stuart Berg
• Move dask/array/top.py code to dask/blockwise.py (GH#4348) Matthew Rocklin
• Add has_parallel_type (GH#4395) Matthew Rocklin
• CI: Update Appveyor (GH#4381) Tom Augspurger
• Ignore non-readable config files (GH#4388) Jim Crist

4.28.10 1.0.0 / 2018-11-28

Array

• Add nanocumsum/nanocumprod unit tests (GH#4215) Guido Imperiale
DataFrame

- Add index to to_dask_dataframe docstring (GH#4232) James Bourbeau
- Text and fix when appending categoricals with fastparquet (GH#4245) Martin Durant
- Don’t reread metadata when passing ParquetFile to read_parquet (GH#4247) Martin Durant

Documentation

- Copy edit documentation (GH#4222) (GH#4224) (GH#4228) (GH#4231) (GH#4230) (GH#4234) (GH#4235) (GH#4254) Miguel Farrajota
- Updated doc for the new scheduler keyword (GH#4251) @milesial

Core

- Avoid a few warnings (GH#4223) Matthew Rocklin
- Remove dask.store module (GH#4221) Matthew Rocklin
- Remove AUTHORS.md Jim Crist

4.28.11 0.20.2 / 2018-11-15

Array

- Avoid fusing dependencies of atop reductions (GH#4207) Matthew Rocklin

Dataframe

- Improve memory footprint for dataframe correlation (GH#4193) Damien Garaud
- Add empty DataFrame check to boundary_slice (GH#4212) James Bourbeau

Documentation

- Copy edit documentation (GH#4197) (GH#4204) (GH#4198) (GH#4199) (GH#4200) (GH#4202) (GH#4209) Miguel Farrajota
- Add stats module namespace (GH#4206) James Bourbeau
- Fix link in dataframe documentation (GH#4208) James Bourbeau

4.28.12 0.20.1 / 2018-11-09

Array

- Only allocate the result space in wrapped_pad_func (GH#4153) John A Kirkham
- Generalize expand_pad_width to expand_pad_value (GH#4150) John A Kirkham
- Test da.pad with 2D linear_ramp case (GH#4162) John A Kirkham
- Fix import for broadcast_to. (GH#4168) samc0de
• Rewrite Dask Array’s pad to add only new chunks (GH#4152) John A Kirkham
• Validate index inputs to atop (GH#4182) Matthew Rocklin

Core
• Dask.config set and get normalize underscores and hyphens (GH#4143) James Bourbeau
• Only subs on core collections, not subclasses (GH#4159) Matthew Rocklin
• Add block_size=0 option to HTTPFileSystem. (GH#4171) Martin Durant
• Add traverse support for dataclasses (GH#4165) Armin Berres
• Avoid optimization on sharedicts without dependencies (GH#4181) Matthew Rocklin
• Update the pytest version for TravisCI (GH#4189) Damien Garaud
• Use key_split rather than funcname in visualize names (GH#4160) Matthew Rocklin

Dataframe
• Add fix for DataFrame.__setitem__ for index (GH#4151) Anderson Banihirwe
• Fix column choice when passing list of files to fastparquet (GH#4174) Martin Durant
• Pass engine_kwargs from read_sql_table to sqlalchem (GH#4187) Damien Garaud

Documentation
• Fix documentation in Delayed best practices example that returned an empty list (GH#4147) Jonathan Fraine
• Copy edit documentation (GH#4164) (GH#4175) (GH#4185) (GH#4192) (GH#4191) (GH#4190) (GH#4180) Miguel Farrajota
• Fix typo in docstring (GH#4183) Carlos Valiente

4.28.13 0.20.0 / 2018-10-26
Array
• Fuse Atop operations (GH#3998), (GH#4081) Matthew Rocklin
• Support da.asanyarray on dask dataframes (GH#4080) Matthew Rocklin
• Remove unnecessary endianness check in datetime test (GH#4113) Elliott Sales de Andrade
• Set name=False in array foo_like functions (GH#4116) Matthew Rocklin
• Remove dask.array.ghost module (GH#4121) Matthew Rocklin
• Fix use of getargspec in dask array (GH#4125) Stephan Hoyer
• Adds dask.array.invert (GH#4127), (GH#4131) Anderson Banihirwe
• Raise informative error on arg-reduction on unknown chunksize (GH#4128), (GH#4135) Matthew Rocklin
• Normalize reversed slices in dask array (GH#4126) Matthew Rocklin
Bag

- Add bag.to_avro (GH#4076) Martin Durant

Core

- Pull num_workers from config.get (GH#4086), (GH#4093) James Bourbeau
- Fix invalid escape sequences with raw strings (GH#4112) Elliott Sales de Andrade
- Raise an error on the use of the get= keyword and set_options (GH#4077) Matthew Rocklin
- Add import for Azure DataLake storage, and add docs (GH#4132) Martin Durant
- Avoid collections.Mapping/Sequence (GH#4138) Matthew Rocklin

Dataframe

- Include index keyword in to_dask_dataframe (GH#4071) Matthew Rocklin
- add support for duplicate column names (GH#4087) Jan Koch
- Implement min_count for the DataFrame methods sum and prod (GH#4090) Bart Broere
- Remove pandas warnings in concat (GH#4095) Matthew Rocklin
- DataFrame.to_csv header option to only output headers in the first chunk (GH#3909) Rahul Vaidya
- Remove Series.to_parquet (GH#4104) Justin Dennison
- Avoid warnings and deprecated pandas methods (GH#4115) Matthew Rocklin
- Swap ‘old’ and ‘previous’ when reporting append error (GH#4130) Martin Durant

Documentation

- Copy edit documentation (GH#4073), (GH#4074), (GH#4094), (GH#4097), (GH#4107), (GH#4124), (GH#4133), (GH#4139) Miguel Farrajota
- Fix typo in code example (GH#4089) Antonino Ingargiola
- Add pycon 2018 presentation (GH#4102) Javad
- Quick description for gcsfs (GH#4109) Martin Durant
- Fixed typo in docstrings of read_sql_table method (GH#4114) TakaakiFuruse
- Make target directories in redirects if they don’t exist (GH#4136) Matthew Rocklin

4.28.14  0.19.4 / 2018-10-09

Array

- Implement apply_gufunc(..., axes=..., keepdims=...) (GH#3985) Markus Gonser

Bag

- Fix typo in datasets.make_people (GH#4069) Matthew Rocklin
Dataframe

- Added percentiles options for dask.dataframe.describe method (GH#4067) Zhenqing Li
- Add DataFrame.partitions accessor similar to Array.blocks (GH#4066) Matthew Rocklin

Core

- Pass get functions and Clients through scheduler keyword (GH#4062) Matthew Rocklin

Documentation

- Fix Typo on hpc example. (missing = in kwarg). (GH#4068) Matthias Bussonier
- Extensive copy-editing: (GH#4065), (GH#4064), (GH#4063) Miguel Farrajota

4.28.15 0.19.3 / 2018-10-05

Array

- Make da.RandomState extensible to other modules (GH#4041) Matthew Rocklin
- Support unknown dims in ravel no-op case (GH#4055) Jim Crist
- Add basic infrastructure for cupy (GH#4019) Matthew Rocklin
- Avoid asarray and lock arguments for from_array(item) (GH#4044) Matthew Rocklin
- Move local imports in corcoef to global imports (GH#4030) John A Kirkham
- Move local indices import to global import (GH#4029) John A Kirkham
- Fix-up Dask Array’s fromfunction w.r.t. dtype and kwarg (GH#4028) John A Kirkham
- Don’t use dummy expansion for trim_internal in overlapped (GH#3964) Mark Harfouche
- Add unravel_index (GH#3958) John A Kirkham

Bag

- Sort result in Bag.frequencies (GH#4033) Matthew Rocklin
- Add support for npartitions=1 edge case in groupby (GH#4050) James Bourbeau
- Add new random dataset for people (GH#4018) Matthew Rocklin
- Improve performance of bag.read_text on small files (GH#4013) Eric Wolak
- Add bag.read_avro (GH#4000) (GH#4007) Martin Durant

Dataframe

- Added an index parameter to dask.dataframe.from_dask_array() for creating a dask DataFrame from a dask Array with a given index. (GH#3991) Tom Augspurger
- Improve sub-classability of dask dataframe (GH#4015) Matthew Rocklin
- Fix failing hdfs test [test-hdfs] (GH#4046) Jim Crist
• fuse_subgraphs works without normal fuse (GH#4042) Jim Crist
• Make path for reading many parquet files without prescan (GH#3978) Martin Durant
• Index in dd.from_dask_array (GH#3991) Tom Augspurger
• Making skiprows accept lists (GH#3975) Julia Signell
• Fail early in fastparquet read for nonexistent column (GH#3989) Martin Durant

Core

• Add support for npartitions=1 edge case in groupby (GH#4050) James Bourbeau
• Automatically wrap large arguments with dask.delayed in map_blocks/partitions (GH#4002) Matthew Rocklin
• Fuse linear chains of subgraphs (GH#3979) Jim Crist
• Make multiprocessing context configurable (GH#3763) Itamar Turner-Trauring

Documentation

• Extensive copy-editing (GH#4049), (GH#4034), (GH#4031), (GH#4020), (GH#4021), (GH#4022), (GH#4023), (GH#4016), (GH#4017), (GH#4010), (GH#3997), (GH#3996), Miguel Farrajota
• Update shuffle method selection docs (GH#4048) James Bourbeau
• Remove docs/source/examples, point to examples.dask.org (GH#4014) Matthew Rocklin
• Replace readthedocs links with dask.org (GH#4008) Matthew Rocklin
• Updates DataFrame.to_hdf docstring for returned values (GH#3992) James Bourbeau

4.28.16 0.19.2 / 2018-09-17

Array

• apply_gufunc implements automatic infer of functions output dtypes (GH#3936) Markus Gonser
• Fix array histogram range error when array has nans (GH#3980) James Bourbeau
• Issue 3937 follow up, int type checks. (GH#3956) Yu Feng
• from_array: add @martindurant’s explaining of how hashing is done for an array. (GH#3965) Mark Harfouche
• Support gradient with coordinate (GH#3949) Keisuke Fujii

Core

• Fix use of has_keyword with partial in Python 2.7 (GH#3966) Mark Harfouche
• Set pyarrow as default for HDFS (GH#3957) Matthew Rocklin
4.28.17 0.19.1 / 2018-09-06

Array

- Don’t enforce dtype if result has no dtype (GH#3928) Matthew Rocklin
- Fix NumPy issubtype deprecation warning (GH#3939) Bruce Merry
- Fix arg reduction tokens to be unique with different arguments (GH#3955) Tobias de Jong
- Coerce numpy integers to ints in slicing code (GH#3944) Yu Feng
- Linalg.norm ndim along axis partial fix (GH#3933) Tobias de Jong

Dataframe

- Deterministic DataFrame.set_index (GH#3867) George Sakkis
- Fix divisions in read_parquet when dealing with filters #3831 #3930 (GH#3923) (GH#3931) @andrethrill
- Fixing returning type in categorical.as_known (GH#3888) Sriharsha Hatwar
- Fix DataFrame.assign for callables (GH#3919) Tom Augspurger
- Include partitions with no width in repartition (GH#3941) Matthew Rocklin
- Don’t constrict stage/k dtype in dataframe shuffle (GH#3942) Matthew Rocklin

Documentation

- DOC: Add hint on how to render task graphs horizontally (GH#3922) Uwe Korn
- Add try-now button to main landing page (GH#3924) Matthew Rocklin

4.28.18 0.19.0 / 2018-08-29

Array

- Support coordinate in gradient (GH#3949) Keisuke Fujii
- Fix argtopk split_every bug (GH#3810) Guido Imperiale
- Ensure result computing dask.array.isnull() always gives a numpy array (GH#3825) Stephan Hoyer
- Support concatenate for scipy.sparse in dask array (GH#3836) Matthew Rocklin
- Fix argtopk on 32-bit systems. (GH#3823) Elliott Sales de Andrade
- Normalize keys in rechunk (GH#3820) Matthew Rocklin
- Allow shape of dask.array to be a numpy array (GH#3844) Mark Harfouche
• Fix numpy deprecation warning on tuple indexing (GH#3851) Tobias de Jong
• Rename ghost module to overlap (GH#3830) Robert Sare
• Re-add the ghost import to da __init__ (GH#3861) Jim Crist
• Ensure copy preserves masked arrays (GH#3852) Tobias de Jong

DataFrame
• Added dtype and sparse keywords to dask.dataframe.get_dummies() (GH#3792) Tom Augspurger
• Added dask.dataframe.to_dask_array() for converting a Dask Series or DataFrame to a Dask Array, possibly with known chunk sizes (GH#3884) Tom Augspurger
• Changed the behavior for dask.array.asarray() for dask dataframe and series inputs. Previously, the series was eagerly converted to an in-memory NumPy array before creating a dask array with known chunks sizes. This caused unexpectedly high memory usage. Now, no intermediate NumPy array is created, and a Dask array with unknown chunk sizes is returned (GH#3884) Tom Augspurger
• DataFrame.iloc (GH#3805) Tom Augspurger
• When reading multiple paths, expand globs. (GH#3828) Irina Truong
• Added index column name after resample (GH#3833) Eric Bonfadini
• Add (lazy) shape property to dataframe and series (GH#3212) Henrique Ribeiro
• Fix failing hdfs test [test-hdfs] (GH#3858) Jim Crist
• Fixes for pyarrow 0.10.0 release (GH#3860) Jim Crist
• Rename to_csv keys for diagnostics (GH#3890) Matthew Rocklin
• Match pandas warnings for concat sort (GH#3897) Tom Augspurger
• Include filename in read_csv (GH#3908) Julia Signell

Core
• Better error message on import when missing common dependencies (GH#3771) Danilo Horta
• Drop Python 3.4 support (GH#3840) Jim Crist
• Remove expired deprecation warnings (GH#3841) Jim Crist
• Add DASK_ROOT_CONFIG environment variable (GH#3849) Joe Hamman
• Don’t cull in local scheduler, do cull in delayed (GH#3856) Jim Crist
• Increase conda download retries (GH#3857) Jim Crist
• Add python_requires and Trove classifiers (GH#3855) @hugovk
• Fix collections.abc deprecation warnings in Python 3.7.0 (GH#3876) Jan Margeta
• Allow dot jpeg to xfail in visualize tests (GH#3896) Matthew Rocklin
• Add Python 3.7 to travis.yml (GH#3894) Matthew Rocklin
• Add expand_environment_variables to dask.config (GH#3893) Joe Hamman
Docs

- Fix typo in import statement of diagnostics (GH#3826) John Mrziglod
- Add link to YARN docs (GH#3838) Jim Crist
- Fix of minor typos in landing page index.html (GH#3746) Christoph Moehl
- Update delayed-custom.rst (GH#3850) Anderson Banihirwe
- DOC: clarify delayed docstring (GH#3709) Scott Sievert
- Add new presentations (GH#3880) Javad
- Add dask array normalize_chunks to documentation (GH#3878) Daniel Rothenberg
- Docs: Fix link to snakeviz (GH#3900) Hans Moritz Günther
- Add missing ‘ to docstring (GH#3915) @rtobar

4.28.19 0.18.2 / 2018-07-23

Array

- Reimplemented argtopk to make it release the GIL (GH#3610) Guido Imperiale
- Don’t overlap on non-overlapped dimensions in map_overlap (GH#3653) Matthew Rocklin
- Fix linalg.tsqr for dimensions of uncertain length (GH#3662) Jeremy Chen
- Break apart uneven array-of-int slicing to separate chunks (GH#3648) Matthew Rocklin
- Align auto chunks to provided chunks, rather than shape (GH#3679) Matthew Rocklin
- Adds endpoint and retstep support for linspace (GH#3675) James Bourbeau
- Implement .blocks accessor (GH#3689) Matthew Rocklin
- Add block_info keyword to map_blocks functions (GH#3686) Matthew Rocklin
- Slice by dask array of ints (GH#3407) Guido Imperiale
- Support dtype in arange (GH#3722) Guido Imperiale
- Fix argtopk with uneven chunks (GH#3720) Guido Imperiale
- Raise error when replace=False in da.choice (GH#3765) James Bourbeau
- Update chunks in Array.__setitem__ (GH#3767) Itamar Turner-Trauring
- Add a chunksize convenience property (GH#3777) Jacob Tomlinson
- Fix and simplify array slicing behavior when step < 0 (GH#3702) Ziyao Wei
- Ensure to_zarr with return_stored True returns a Dask Array (GH#3786) John A Kirkham

Bag

- Add last_endline optional parameter in to_textfiles (GH#3745) George Sakkis
Dataframe

- Add aggregate function for rolling objects (GH#3772) Gerome Pistre
- Properly tokenize cumulative groupby aggregations (GH#3799) Cloves Almeida

Delayed

- Add the @ operator to the delayed objects (GH#3691) Mark Harfouche
- Add delayed best practices to documentation (GH#3737) Matthew Rocklin
- Fix @delayed decorator for methods and add tests (GH#3757) Ziyao Wei

Core

- Fix extra progressbar (GH#3669) Mike Neish
- Allow tasks back onto ordering stack if they have one dependency (GH#3652) Matthew Rocklin
- Prefer end-tasks with low numbers of dependencies when ordering (GH#3588) Tom Augspurger
- Add assert_eq to top-level modules (GH#3726) Matthew Rocklin
- Test that dask collections can hold scipy.sparse arrays (GH#3738) Matthew Rocklin
- Fix setup of lz4 decompression functions (GH#3782) Elliott Sales de Andrade
- Add datasets module (GH#3780) Matthew Rocklin

4.28.20 0.18.1 / 2018-06-22

Array

- from_array now supports scalar types and nested lists/tuples in input, just like all numpy functions do; it also produces a simpler graph when the input is a plain ndarray (GH#3568) Guido Imperiale
- Fix slicing of big arrays due to cumsum dtype bug (GH#3620) Marco Rossi
- Add Dask Array implementation of pad (GH#3578) John A Kirkham
- Fix array random API examples (GH#3625) James Bourbeau
- Add average function to dask array (GH#3640) James Bourbeau
- Tokenize ghost_internal with axes (GH#3643) Matthew Rocklin
- Add outer for Dask Arrays (GH#3658) John A Kirkham

DataFrame

- Add Index.to_series method (GH#3613) Henrique Ribeiro
- Fix missing partition columns in pyarrow-parquet (GH#3636) Martin Durant
Core

- Minor tweaks to CI (GH#3629) Guido Imperiale
- Add back dask.utils.effective_get (GH#3642) Matthew Rocklin
- DASK_CONFIG dictates config write location (GH#3621) Jim Crist
- Replace ‘collections’ key in unpack_collections with unique key (GH#3632) Yu Feng
- Avoid deepcopy in dask.config.set (GH#3649) Matthew Rocklin

4.28.21 0.18.0 / 2018-06-14

Array

- Add to/from_zarr for Zarr-format datasets and arrays (GH#3460) Martin Durant
- Experimental addition of generalized ufunc support, apply_gufunc, gufunc, and as_gufunc (GH#3109) (GH#3526) (GH#3539) Markus Gonser
- Avoid unnecessary rechunking tasks (GH#3529) Matthew Rocklin
- Compute dtypes at runtime for fft (GH#3511) Matthew Rocklin
- Generate UUIDs for all da.store operations (GH#3540) Martin Durant
- Correct internal dimension of Dask’s SVD (GH#3517) John A Kirkham
- BUG: do not raise IndexError for identity slice in array.vindex (GH#3559) Scott Sievert
- Adds isneginf and isposinf (GH#3581) John A Kirkham
- Drop Dask Array’s learn module (GH#3580) John A Kirkham
- added sfqr (short-and-fat) as a counterpart to tsqr... (GH#3575) Jeremy Chen
- Allow 0-width chunks in dask.array.rechunk (GH#3591) Marc Pfister
- Document Dask Array’s nan_to_num in public API (GH#3599) John A Kirkham
- Show block example (GH#3601) John A Kirkham
- Replace token= keyword with name= in map_blocks (GH#3597) Matthew Rocklin
- Disable locking in to_zarr (needed for using to_zarr in a distributed context) (GH#3607) John A Kirkham
- Support Zarr Arrays in to_zarr/from_zarr (GH#3561) John A Kirkham
- Added recursion to array/linalg/tsqr to better manage the single core bottleneck (GH#3586) Jeremy Chan (GH#3396) Guido Imperiale

Dataframe

- Add to/read_json (GH#3494) Martin Durant
- Adds index to unsupported arguments for DataFrame.rename method (GH#3522) James Bourbeau
- Adds support to subset Dask DataFrame columns using numpy.ndarray, pandas.Series, and pandas.Index objects (GH#3536) James Bourbeau
- Raise error if meta columns do not match dataframe (GH#3485) Christopher Ren
- Add index to unsupported argument for DataFrame.rename (GH#3522) James Bourbeau
• Adds support for subsetting DataFrames with pandas Index/Series and numpy ndarrays (GH#3536) James Bourbeau
• Dataframe sample method docstring fix (GH#3566) James Bourbeau
• fixes dd.read_json to infer file compression (GH#3594) Matt Lee
• Adds n to sample method (GH#3606) James Bourbeau
• Add fastparquet ParquetFile object support (GH#3573) @andrethrill

Bag

• Rename method= keyword to shuffle= in bag.groupby (GH#3470) Matthew Rocklin

Core

• Replace get= keyword with scheduler= keyword (GH#3448) Matthew Rocklin
• Add centralized dask.config module to handle configuration for all Dask subprojects (GH#3432) (GH#3513) (GH#3520) Matthew Rocklin
• Add dask-ssh CLI Options and Description. (GH#3476) @beomi
• Read whole files fix regardless of header for HTTP (GH#3496) Martin Durant
• Adds synchronous scheduler syntax to debugging docs (GH#3509) James Bourbeau
• Replace dask.set_options with dask.config.set (GH#3502) Matthew Rocklin
• Update sphinx readthedocs-theme (GH#3516) Matthew Rocklin
• Introduce “auto” value for normalize_chunks (GH#3507) Matthew Rocklin
• Fix check in configuration with env=None (GH#3562) Simon Perkins
• Update sizeof definitions (GH#3582) Matthew Rocklin
• Remove –verbose flag from travis-ci (GH#3477) Matthew Rocklin
• Remove “da.random” from random array keys (GH#3604) Matthew Rocklin

4.28.22 0.17.5 / 2018-05-16

Array

• Fix rechunk with chunksize of -1 in a dict (GH#3469) Stephan Hoyer
• einsum now accepts the split_every parameter (GH#3471) Guido Imperiale
• Improved slicing performance (GH#3479) Yu Feng

DataFrame

• Compatibility with pandas 0.23.0 (GH#3499) Tom Augspurger
Dask Documentation, Release 1.2.2

4.28.23 0.17.4 / 2018-05-03

Dataframe

- Add support for indexing Dask DataFrames with string subclasses (GH#3461) James Bourbeau
- Allow using both sorted_index and chunksize in read_hdf (GH#3463) Pierre Bartet
- Pass filesystem to arrow piece reader (GH#3466) Martin Durant
- Switches to using dask.compat string_types (GH#3462) James Bourbeau

4.28.24 0.17.3 / 2018-05-02

Array

- Add einsum for Dask Arrays (GH#3412) Simon Perkins
- Add piecewise for Dask Arrays (GH#3350) John A Kirkham
- Fix handling of nan in broadcast_shapes (GH#3356) John A Kirkham
- Add isin for dask arrays (GH#3363), Stephan Hoyer
- Overhauled topk for Dask Arrays: faster algorithm, particularly for large k's; added support for multiple axes, recursive aggregation, and an option to pick the bottom k elements instead. (GH#3395) Guido Imperiale
- The topk API has changed from topk(k, array) to the more conventional topk(array, k). The legacy API still works but is now deprecated. (GH#2965) Guido Imperiale
- New function argtopk for Dask Arrays (GH#3396) Guido Imperiale
- Fix handling partial depth and boundary in map_overlap (GH#3445) John A Kirkham
- Add gradient for Dask Arrays (GH#3434) John A Kirkham

Dataframe

- Allow t as shorthand for table in to_hdf for pandas compatibility (GH#3330) Jörg Dietrich
- Added top level isna method for Dask DataFrames (GH#3294) Christopher Ren
- Fix selection on partition column on read_parquet for engine="pyarrow" (GH#3207) Uwe Korn
- Added DataFrame.squeeze method (GH#3366) Christopher Ren
- Added infer_divisions option to read_parquet to specify whether read engines should compute divisions (GH#3387) Jon Mease
- Added support for inferring division for engine="pyarrow" (GH#3387) Jon Mease
- Provide more informative error message for meta= errors (GH#3343) Matthew Rocklin
- add orc reader (GH#3284) Martin Durant
- Default compression for parquet now always Snappy, in line with pandas (GH#3373) Martin Durant
- Fixed bug in Dask DataFrame and Series comparisons with NumPy scalars (GH#3436) James Bourbeau
- Remove outdated requirement from repartition docstring (GH#3440) Jörg Dietrich
- Fixed bug in aggregation when only a Series is selected (GH#3446) Jörg Dietrich

4.28. Changelog 867
• Add default values to make_timeseries (GH#3421) Matthew Rocklin

Core

• Support traversing collections in persist, visualize, and optimize (GH#3410) Jim Crist
• Add schedule= keyword to compute and persist. This replaces common use of the get= keyword (GH#3448) Matthew Rocklin

4.28.25 0.17.2 / 2018-03-21

Array

• Add broadcast_arrays for Dask Arrays (GH#3217) John A Kirkham
• Add bitwise_* ufuncs (GH#3219) John A Kirkham
• Add optional axis argument to squeeze (GH#3261) John A Kirkham
• Validate inputs to atop (GH#3307) Matthew Rocklin
• Avoid calls to astype in concatenate if all parts have the same dtype (GH#3301) Martin Durant

DataFrame

• Fixed bug in shuffle due to aggressive truncation (GH#3201) Matthew Rocklin
• Support specifying categorical columns on read_parquet with categories=[...], for engine="pyarrow" (GH#3177) Uwe Korn
• Add dd.tseries.Resampler.agg (GH#3202) Richard Postelnik
• Support operations that mix dataframes and arrays (GH#3230) Matthew Rocklin
• Support extra Scalar and Delayed args in dd.groupby._Groupby.apply (GH#3256) Gabriele Lanaro

Bag

• Support joining against single-partitioned bags and delayed objects (GH#3254) Matthew Rocklin

Core

• Fixed bug when using unexpected but hashable types for keys (GH#3238) Daniel Collins
• Fix bug in task ordering so that we break ties consistently with the key name (GH#3271) Matthew Rocklin
• Avoid sorting tasks in order when the number of tasks is very large (GH#3298) Matthew Rocklin

4.28.26 0.17.1 / 2018-02-22

Array

• Corrected dimension chunking in indices (GH#3166, GH#3167) Simon Perkins
• Inline store_chunk calls for store's return_stored option (GH#3153) John A Kirkham
• Compatibility with struct dtypes for NumPy 1.14.1 release (GH#3187) Matthew Rocklin

**DataFrame**

• Bugfix to allow column assignment of pandas datetimes(GH#3164) Max Epstein

**Core**

• New file-system for HTTP(S), allowing direct loading from specific URLs (GH#3160) Martin Durant
• Fix bug when tokenizing partials with no keywords (GH#3191) Matthew Rocklin
• Use more recent LZ4 API (GH#3157) Thrasibule
• Introduce output stream parameter for progress bar (GH#3185) Dieter Weber

**4.28.27 0.17.0 / 2018-02-09**

**Array**

• Added a support object-type arrays for nansum, nanmin, and nanmax (GH#3133) Keisuke Fujii
• 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
• Support arrays with unknown chunk sizes in percentile (GH#3107) Matthew Rocklin
• Tokenize scipy.sparse arrays and np.matrix (GH#3060) Roman Yurchak

**DataFrame**

• Support month timedeltas in repartition(freq=...) (GH#3110) Matthew Rocklin
• Avoid mutation in dataframe groupby tests (GH#3118) Matthew Rocklin
• read_csv, read_table, and read_parquet accept iterables of paths (GH#3124) Jim Crist
• Deprecates the dd.to_delayed function in favor of the existing method (GH#3126) Jim Crist
• Return dask.arrays from df.map_partitions calls when the UDF returns a numpy array (GH#3147) Matthew Rocklin
• Change handling of columns and index in dd.read_parquet to be more consistent, especially in handling of multi-indices (GH#3149) Jim Crist
• fastparquet append=True allowed to create new dataset (GH#3097) Martin Durant
• dtype rationalization for sql queries (GH#3100) Martin Durant

**Bag**

• Document bag.map_partitions function may recieve either a list or generator. (GH#3150) Nir
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) (GH#3122) 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
• Change task ordering to do a full traversal (GH#3066) Matthew Rocklin
• Adds an optimize_graph keyword to all to_delayed methods to allow controlling whether optimizations occur on conversion. (GH#3126) Jim Crist
• Support using pyarrow for hdfs integration (GH#3123) Jim Crist
• Move HDFS integration and tests into dask repo (GH#3083) Jim Crist
• Remove write_bytes (GH#3116) Jim Crist

4.28.28 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
• Compatability 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` (GH#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

4.28.29 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 type 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) Jesse 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 sharedict keys (GH#2763) Matthew Rocklin
• Calling compute within a dask.distributed task defaults to distributed scheduler (GH#2762) Matthew Rocklin
• Auto-import gcsfs when gs:// 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

4.28.30 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_npy_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` (GH#2710)

Core

• Drop `s3fs` from `pip install dask[complete]` (GH#2750)

4.28.31 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)

4.28.32 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)

4.28.33 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)

4.28.34 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)

4.28.35 0.14.3 / 2017-05-05

DataFrame

- Pandas 0.20.0 support

4.28.36 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)
- Tensordot 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)

4.28.37  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)

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)
4.28.38 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)
4.28.39 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)
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)

4.28.41 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
4.28.42 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

4.28.43 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. The 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.

4.28.44 0.9.0 / 2016-05-11

API Changes

- \texttt{dask.do} and \texttt{dask.value} have been renamed to \texttt{dask.delayed}
- \texttt{dask.bag.from_filenames} has been renamed to \texttt{dask.bag.read_text}
- All S3/HDFS data ingest functions like \texttt{db.from_s3} or \texttt{distributed.s3.read_csv} have been moved into the plain \texttt{read_text}, \texttt{read_csv} functions, which now support protocols, like \texttt{dd.read_csv('s3://bucket/keys*.csv')}.

Array

- Add support for \texttt{scipy.LinearOperator}
- Improve optional locking to on-disk data structures
- Change rechunk to expose the intermediate chunks

Bag

- Rename \texttt{from_filenames} to \texttt{read_text}
- Remove \texttt{from_s3} in favor of \texttt{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 unsplittable compression types, like gzip or bz2
- Improve hashing for single-machine shuffle operations
- Add new callback method for start state
- General performance tuning

4.28.45 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

4.28.46 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

4.28.47 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

4.28.48 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.dask.org. General feedback on that project is most welcome from this community.

4.28.49 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`.

4.28.50 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

4.28.51 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

4.29 Configuration

Taking full advantage of Dask sometimes requires user configuration. This might be to control logging verbosity, specify cluster configuration, provide credentials for security, or any of several other options that arise in production.

Configuration is specified in one of the following ways:

1. YAML files in ~/.config/dask/ or /etc/dask/
2. Environment variables like DASK_DISTRIBUTED__SCHEDULER__WORK_STEALING=True
3. Default settings within sub-libraries

This combination makes it easy to specify configuration in a variety of settings ranging from personal workstations, to IT-mandated configuration, to docker images.

4.29.1 Access Configuration

```python
from dask import config

# populate config with distributed defaults
>>> dask.config.config

{ 'logging': { 'distributed': 'info', 'bokeh': 'critical', 'tornado': 'critical' }, 'admin': { 'log-format': '%(name)s - %(levelname)s - %(message)s' }}
```

Configuration is usually read by using the `dask.config` module, either with the `config` dictionary or the `get` function:
You may wish to inspect the `dask.config.config` dictionary to get a sense for what configuration is being used by your current system.

Note that the `get` function treats underscores and hyphens identically. For example, `dask.config.get('num_workers')` is equivalent to `dask.config.get('num-workers')`.

### 4.29.2 Specify Configuration

#### YAML files

You can specify configuration values in YAML files like the following:

```
logging:
  distributed: info
  bokeh: critical
  tornado: critical

scheduler:
  work-stealing: True
  allowed-failures: 5

admin:
  log-format: '%(name)s - %(levelname)s - %(message)s'
```

These files can live in any of the following locations:

1. The `~/.config/dask` directory in the user’s home directory
2. The `{sys.prefix}/etc/dask` directory local to Python
3. The root directory (specified by the `DASK_ROOT_CONFIG` environment variable or `/etc/dask/` by default)

Dask searches for all YAML files within each of these directories and merges them together, preferring configuration files closer to the user over system configuration files (preference follows the order in the list above). Additionally, users can specify a path with the `DASK_CONFIG` environment variable, which takes precedence at the top of the list above.

The contents of these YAML files are merged together, allowing different Dask subprojects like `dask-kubernetes` or `dask-ml` to manage configuration files separately, but have them merge into the same global configuration.

*Note: for historical reasons we also look in the ‘`~/.dask`’ directory for config files. This is deprecated and will soon be removed.*

#### Environment Variables

You can also specify configuration values with environment variables like the following:
export DASK_DISTRIBUTED__SCHEDULER__WORK_STEALING=True
export DASK_DISTRIBUTED__SCHEDULER__ALLOWED_FAILURES=5

resulting in configuration values like the following:

```
{'distributed':
 'scheduler':
  {'work-stealing': True,
   'allowed-failures': 5}
}
```

Dask searches for all environment variables that start with DASK_, then transforms keys by converting to lower case and changing double-underscores to nested structures.

Dask tries to parse all values with ast.literal_eval, letting users pass numeric and boolean values (such as True in the example above) as well as lists, dictionaries, and so on with normal Python syntax.

Environment variables take precedence over configuration values found in YAML files.

Defaults

Additionally, individual subprojects may add their own default values when they are imported. These are always added with lower priority than the YAML files or environment variables mentioned above:

```python
>>> import dask.config
>>> dask.config.config # no configuration by default
{}
>>> import dask.distributed
>>> dask.config.config # New values have been added
{'scheduler': ..., 'worker': ..., 'tls': ...}
```

Directly within Python

```
dask.config.set([arg, config, lock]) Temporarily set configuration values within a context manager
```

Configuration is stored within a normal Python dictionary in dask.config.config and can be modified using normal Python operations.

Additionally, you can temporarily set a configuration value using the dask.config.set function. This function accepts a dictionary as an input and interprets "." as nested access:

```python
>>> dask.config.set({'scheduler.work-stealing': True})
```

This function can also be used as a context manager for consistent cleanup:

```
with dask.config.set({'scheduler.work-stealing': True}):
  ...
```

Note that the set function treats underscores and hyphens identically. For example, dask.config.
set({"scheduler.work-stealing": True}) is equivalent to dask.config.set({"scheduler.
work_stealing": True}).

4.29.3 Updating Configuration

Manipulating configuration dictionaries

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dask.config.merge(*dicts)</td>
<td>Update a sequence of nested dictionaries</td>
</tr>
<tr>
<td>dask.config.update(old, new[, priority])</td>
<td>Update a nested dictionary with values from another</td>
</tr>
<tr>
<td>dask.config.expand_environment_variables(config)</td>
<td>Expand environment variables in a nested config dictionary</td>
</tr>
</tbody>
</table>

As described above, configuration can come from many places, including several YAML files, environment variables, and project defaults. Each of these provides a configuration that is possibly nested like the following:

```python
x = {'a': 0, 'c': {'d': 4}}
y = {'a': 1, 'b': 2, 'c': {'e': 5}}
```

Dask will merge these configurations respecting nested data structures, and respecting order:

```python
>>> dask.config.merge(x, y)
{'a': 1, 'b': 2, 'c': {'d': 4, 'e': 5}}
```

You can also use the `update` function to update the existing configuration in place with a new configuration. This can be done with priority being given to either config. This is often used to update the global configuration in `dask.config`:

```python
dask.config.update(dask.config, new, priority='new')  # Give priority to new values
dask.config.update(dask.config, new, priority='old')  # Give priority to old values
```

Sometimes it is useful to expand environment variables stored within a configuration. This can be done with the `expand_environment_variables` function:

```python
dask.config.config = dask.config.expand_environment_variables(dask.config.config)
```

Refreshing Configuration

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dask.config.collect([paths, env])</td>
<td>Collect configuration from paths and environment variables</td>
</tr>
<tr>
<td>dask.config.refresh([config, defaults])</td>
<td>Update configuration by re-reading yaml files and env variables</td>
</tr>
</tbody>
</table>

If you change your environment variables or YAML files, Dask will not immediately see the changes. Instead, you can call `refresh` to go through the configuration collection process and update the default configuration:

```python
>>> dask.config.config
{}

>>> # make some changes to yaml files

>>> dask.config.refresh()
```

(continues on next page)
This function uses `dask.config.collect`, which returns the configuration without modifying the global configuration. You might use this to determine the configuration of particular paths not yet on the config path:

```python
>>> dask.config.collect(paths=[])  
 [...]
```

## 4.29.4 Downstream Libraries

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dask.config.ensure_file(source[, ...])</code></td>
<td>Copy file to default location if it does not already exist</td>
</tr>
<tr>
<td><code>dask.config.update(old, new[, priority])</code></td>
<td>Update a nested dictionary with values from another</td>
</tr>
<tr>
<td><code>dask.config.update_defaults(new[, config, ...])</code></td>
<td>Add a new set of defaults to the configuration</td>
</tr>
</tbody>
</table>

Downstream Dask libraries often follow a standard convention to use the central Dask configuration. This section provides recommendations for integration using a fictional project, `dask-foo`, as an example.

Downstream projects typically follow the following convention:

1. Maintain default configuration in a YAML file within their source directory:

```python
setup.py
dask_foo/__init__.py
dask_foo/config.py
dask_foo/core.py
dask_foo/foo.yaml  # <---
```

2. Place configuration in that file within a namespace for the project:

```yaml
# dask_foo/foo.yaml

foo:
  color: red
  admin:
    a: 1
    b: 2
```

3. Within a config.py file (or anywhere) load that default config file and update it into the global configuration:

```python
# dask_foo/config.py
import os
import yaml
import dask.config

fn = os.path.join(os.path.dirname(__file__), '__file__', 'foo.yaml')

with open(fn) as f:
    defaults = yaml.load(f)
dask.config.update_defaults(defaults)
```
4. Within that same config.py file, copy the 'foo.yaml' file to the user's configuration directory if it doesn’t already exist.

We also comment the file to make it easier for us to change defaults in the future.

```python
# ... continued from above

dask.config.ensure_file(source=fn, comment=True)
```

The user can investigate `~/.config/dask/*.yaml` to see all of the commented out configuration files to which they have access.

5. Ensure that this file is run on import by including it in __init__.py:

```python
# dask_foo/__init__.py
from . import config
```

6. Within dask_foo code, use the dask.config.get function to access configuration values:

```python
# dask_foo/core.py

def process(fn, color=dask.config.get('foo.color')):
    ...
```

7. You may also want to ensure that your yaml configuration files are included in your package. This can be accomplished by including the following line in your MANIFEST.in:

```plaintext
recursive-include <PACKAGE_NAME> *.yaml
```

and the following in your setup.py setup call:

```python
from setuptools import setup

setup(...,
       include_package_data=True,
       ...)
```

This process keeps configuration in a central place, but also keeps it safe within namespaces. It places config files in an easy to access location by default (`~/.config/dask/*.yaml`), so that users can easily discover what they can change, but maintains the actual defaults within the source code, so that they more closely track changes in the library.

However, downstream libraries may choose alternative solutions, such as isolating their configuration within their library, rather than using the global dask.config system. All functions in the dask.config module also work with parameters, and do not need to mutate global state.

### 4.29.5 API

```python
dask.config.get (key, default='__no_default__', config={'array': {'chunk-size': '128MiB', 'rechunk-threshold': 4}, 'temporary-directory': None})
```

Get elements from global config

Use '.' for nested access

See also:

```python
dask.config.set
```
Examples

```python
>>> from dask import config
>>> config.get('foo')  # doctest: +SKIP
{'x': 1, 'y': 2}

>>> config.get('foo.x')  # doctest: +SKIP
1

>>> config.get('foo.x.y', default=123)  # doctest: +SKIP
123
```

dask.config.set(arg=None, config={'array': {'chunk-size': '128MiB', 'rechunk-threshold': 4}, 'temporary-directory': None}, lock=<unlocked _thread.lock object>, **kwargs)

Temporarily set configuration values within a context manager

See also:
dask.config.get

Examples

```python
>>> import dask
>>> with dask.config.set({'foo': 123}):
...   pass
```

dask.config.merge(*dicts)

Update a sequence of nested dictionaries

This prefers the values in the latter dictionaries to those in the former

See also:
dask.config.update

Examples

```python
>>> a = {'x': 1, 'y': {'a': 2}}
>>> b = {'y': {'b': 3}}
>>> merge(a, b)  # doctest: +SKIP
{'x': 1, 'y': {'a': 2, 'b': 3}}
```

dask.config.update(old, new, priority='new')

Update a nested dictionary with values from another

This is like dict.update except that it smoothly merges nested values

This operates in-place and modifies old

Parameters

priority: string `{'old', 'new'}` If new (default) then the new dictionary has preference. Otherwise the old dictionary does.

See also:
dask.config.merge
Examples

```python
>>> a = {'x': 1, 'y': {'a': 2}}
>>> b = {'x': 2, 'y': {'b': 3}}
>>> update(a, b)  # doctest: +SKIP
{'x': 2, 'y': {'a': 2, 'b': 3}}
```

```python
>>> a = {'x': 1, 'y': {'a': 2}}
>>> b = {'x': 2, 'y': {'b': 3}}
>>> update(a, b, priority='old')  # doctest: +SKIP
{'x': 1, 'y': {'a': 2, 'b': 3}}
```

dask.config.collect

Collect configuration from paths and environment variables

**Parameters**

- `paths` [List[str]] A list of paths to search for yaml config files
- `env` [dict] The system environment variables

**Returns**

- `config`: dict

**See also:**

dask.config.refresh

collect configuration and update into primary config

```python
dask.config.refresh(config={'array': {'chunk-size': '128MiB', 'rechunk-threshold': 4}, 'temporary-directory': None}, defaults=[{'temporary-directory': None}, {'array': {'chunk-size': '128MiB', 'rechunk-threshold': 4}}], **kwargs)
```

**Update configuration by re-reading yaml files and env variables**

This mutates the global dask.config.config, or the config parameter if passed in.

This goes through the following stages:

1. Clearing out all old configuration
2. Updating from the stored defaults from downstream libraries (see update_defaults)
3. Updating from yaml files and environment variables

Note that some functionality only checks configuration once at startup and may not change behavior, even if configuration changes. It is recommended to restart your python process if convenient to ensure that new configuration changes take place.

**See also:**

dask.config.collect for parameters

dask.config.update_defaults

dask.config.ensure_file

Copy file to default location if it does not already exist

This tries to move a default configuration file to a default location if it does not already exist. It also comments out that file by default.
This is to be used by downstream modules (like dask.distributed) that may have default configuration files that they wish to include in the default configuration path.

**Parameters**

- **source** [string, filename] Source configuration file, typically within a source directory.
- **destination** [string, directory] Destination directory. Configurable by DASK_CONFIG environment variable, falling back to ~/.config/dask.
- **comment** [bool, True by default] Whether or not to comment out the config file when copying.

```python
dask.config.expand_environment_variables(config)
```

Expand environment variables in a nested config dictionary

This function will recursively search through any nested dictionaries and/or lists.

**Parameters**

- **config** [dict, iterable, or str] Input object to search for environment variables

**Returns**

- **config** [same type as input]

**Examples**

```python
>>> expand_environment_variables({'x': [1, 2, '$USER']})
{'x': [1, 2, 'my-username']}
```

### 4.30 Presentations On Dask

- SciPy 2018, July 2018
  - Scalable Machine Learning with Dask (30 minutes)
- PyCon 2018, May 2018
  - Democratizing Distributed Computing with Dask and JupyterHub (32 minutes)
- AMS & ESIP, January 2018
  - Pangeo quick demo: Dask, XArray, Zarr on the cloud with JupyterHub (3 minutes)
  - Pangeo talk: An open-source big data science platform with Dask, XArray, Zarr on the cloud with Jupyter-Hub (43 minutes)
- PYCON.DE 2017, November 2017
  - Dask: Parallelism in Python (1 hour, 2 minutes)
- PYCON 2017, May 2017
  - Dask: A Pythonic Distributed Data Science Framework (46 minutes)
- PLOTCON 2016, December 2016
  - Visualizing Distributed Computations with Dask and Bokeh (33 minutes)
- PyData DC, October 2016
  - Using Dask for Parallel Computing in Python (44 minutes)
• SciPy 2016, July 2016
  – Dask Parallel and Distributed Computing (28 minutes)
• PyData NYC, December 2015
  – Dask Parallelizing NumPy and Pandas through Task Scheduling (33 minutes)
• PyData Seattle, August 2015
  – Dask: out of core arrays with task scheduling (1 hour, 50 minutes)
• SciPy 2015, July 2015
  – Dask Out of core NumPy: Pandas through Task Scheduling (16 minutes)

4.31 Dask Cheat Sheet

The 300KB pdf Dask cheat sheet is a single page summary about using Dask. It is commonly distributed at conferences and trade shows.

4.32 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.

4.32.1 Summary

Generally Dask is smaller and lighter weight than Spark. This means that it has fewer features and, instead, is used in conjunction with other libraries, particularly those in the numeric Python ecosystem. It couples with 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 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, which combines Apache Spark with NumPy arrays.
    – 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.

### 4.32.2 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

### 4.32.3 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 local computing to cluster computing
- You want to interoperate with other technologies and don’t mind installing multiple packages

### 4.32.4 Reasons to choose both

It is easy to use both Dask and Spark on the same data and on the same cluster. They can both read and write common formats, like CSV, JSON, ORC, and Parquet, making it easy to hand results off between Dask and Spark workflows.

They can both deploy on the same clusters. Most clusters are designed to support many different distributed systems at the same time, using resource managers like Kubernetes and YARN. If you already have a cluster on which you run Spark workloads, it’s likely easy to also run Dask workloads on your current infrastructure and vice versa.

In particular, for users coming from traditional Hadoop/Spark clusters (such as those sold by Cloudera/Hortonworks) you are using the Yarn resource manager. You can deploy Dask on these systems using the Dask Yarn project, as well as other projects, like JupyterHub on Hadoop.
4.32.5 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.

4.32.6 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.

4.33 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.

4.33.1 Motivating Example

Consider computing the maximum value of a column in a CSV file:
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:

```python
>>> 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.

### 4.33.2 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:

```python
>>> 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:

```python
>>> 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.

### 4.33.3 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:

```python
>>> 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 \texttt{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 \texttt{df.amount} column will likely be evicted to make space for other more timely results:

\begin{verbatim}
>>> df.amount.max().compute()  # slow the first time
1000
>>> df.amount.min().compute()  # fast because \texttt{df.amount} is in the cache
-1000
>>> df.id.nunique().compute()  # starts to push out \texttt{df.amount} from cache
4.33.4 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 \texttt{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:

\begin{verbatim}
>>> cache.cache.data
<stored values>
>>> cache.cache.heap.heap
<scores of items in cache>
>>> cache.cache.nbytes
<number of bytes per item in cache>
\end{verbatim}

The cache object is powered by \texttt{cachey}, a tiny library for opportunistic caching.

4.33.5 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.

4.34 Internal Data Ingestion

Dask contains internal tools for extensible data ingestion in the \texttt{dask.bytes} package. These functions are developer-focused rather than for direct consumption by users. These functions power user facing functions like \texttt{dd.read_csv} and \texttt{db.read_text} which are probably more useful for most users.
**read_bytes**(urlpath[, delimiter, not_zero, ...])

Given a path or paths, return delayed objects that read from those paths.

**open_files**(urlpath[, mode, compression, ...])

Given a path or paths, return a list of OpenFile 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.

### 4.34.1 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.

### 4.34.2 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*.

### 4.34.3 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.

### 4.34.4 Functions

```
*dask.bytes.read_bytes*(urlpath, delimiter=None, not_zero=False, blocksize='128 MiB', sample='10 kiB', compression=None, include_path=False, **kwargs)
```

Given a path or paths, return delayed objects that read from those paths.
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 or list] Absolute or relative filepath(s). Prefix with a protocol like s3:// to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol.
delimiter [bytes] An optional delimiter, like b'
' on which to split blocks of bytes.
blocksize [int, str] Chunk size in bytes, defaults to "128 MiB"
compression [string or None] String like 'gzip' or 'xz'. Must support efficient random access.
sample [int, string, or boolean] Whether or not to return a header sample. Values can be False for "no sample requested" Or an integer or string value like 2**20 or "1 MiB"
include_path [bool] Whether or not to include the path with the bytes representing a particular file. Default is False.
**kwargs [dict] Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns

sample [bytes] The sample header
blocks [list of lists of dask.Delayed] Each list corresponds to a file, and each delayed object computes to a block of bytes from that file.
paths [list of strings, only included if include_path is True] List of same length as blocks, where each item is the path to the file represented in the corresponding block.

Examples

```python
>>> sample, blocks = read_bytes('2015-*.csv', delimiter=b'
')  # doctest: +SKIP
>>> sample, blocks = read_bytes('s3://bucket/2015-*.csv', delimiter=b'
')  # doctest: +SKIP
>>> sample, paths, blocks = read_bytes('2015-*.csv', include_path=True)  # doctest: +SKIP
```

dask.bytes.open_files (urlpath, mode='rb', compression=None, encoding='utf8', errors=None, name_function=None, num=1, **kwargs)
Given a path or paths, return a list of OpenFile objects.

Parameters

urlpath [string or list] Absolute or relative filepath(s). Prefix with a protocol like s3:// to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol.
mode ['rb', 'wt', etc.]
compression [string] Compression to use. See dask.bytes.compression.files for options.
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 ‘‘OpenFile‘‘ objects.

Examples
>>> files = open_files('2015-*-.csv')  # doctest: +SKIP
>>> files = open_files('s3://bucket/2015-*-.csv.gz', compression='gzip')  # doctest: +SKIP

4.35 Remote Data

Dask can read data from a variety of data stores including local file systems, network file systems, cloud object stores, and Hadoop. Typically this is done by prepending a protocol like "s3://" to paths used in common data access functions like `dd.read_csv`:

```python
import dask.dataframe as dd
df = dd.read_csv('s3://bucket/path/to/data-*.csv')
df = dd.read_parquet('gcs://bucket/path/to/data-*.parq')

import dask.bag as db
b = db.read_text('hdfs://path/to/*.json').map(json.loads)
```

The following remote services are well supported and tested against the main codebase:

- **Local or Network File System**: `file://` - the local file system, default in the absence of any protocol
- **Hadoop File System**: `hdfs://` - Hadoop Distributed File System, for resilient, replicated files within a cluster. Can use either `hdfs3` or `pyarrow`.
- **Amazon S3**: `s3://` - Amazon S3 remote binary store, often used with Amazon EC2, using the library `s3fs`
- **Google Cloud Storage**: `gcs://` or `gs:` - Google Cloud Storage, typically used with Google Compute resource using `gsfs` (in development)
- **HTTP(s)**: `http://` or `https://` for reading data directly from HTTP web servers
- **Azure Datalake Storage**: `adl://`, for use with the Microsoft Azure platform, using `azure-data-lake-store-python`. This is experimental, use at your own risk.

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://`).

Lower-level details on how Dask handles remote data is described in Section *Internal Data Ingestion*.
4.35.1 Optional Parameters

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:

```python
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 back-end 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 to which they refer.

4.35.2 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.

4.35.3 Hadoop File System

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.

HDFS support can be provided by either `hdfs3` or `pyarrow`, defaulting to the first library installed in that order. To explicitly set which library to use, set `hdfs_driver` using `dask.config.set`:

```python
# Use first available option in {hdfs3, pyarrow}
dd.read_csv('hdfs:///path/to/*.csv')

# Use hdfs3 for HDFS I/O
with dask.config.set(hdfs_driver='hdfs3'):
    dd.read_csv('hdfs:///path/to/*.csv')

# Use pyarrow for HDFS I/O
with dask.config.set(hdfs_driver='pyarrow'):
    dd.read_csv('hdfs:///path/to/*.csv')

# Set pyarrow as the global hdfs driver
dask.config.set(hdfs_driver='pyarrow')
```

By default, both libraries attempt 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`. 

906 Chapter 4. Index
Extra Configuration for HDFS3

The following additional options may be passed to the hdfs3 driver via storage_options:

- *host*, *port*, *user*: Basic authentication
- *ticket_cache*, *token*: Kerberos authentication
- *pars*: Dictionary of further parameters (e.g., for high availability)

The hdfs3 driver can also be affected by a few environment variables. For more information on these, see the hdfs3 documentation.

Extra Configuration for PyArrow

The following additional options may be passed to the pyarrow driver via storage_options:

- *host*, *port*, *user*: Basic authentication
- *kerb_ticket*: Path to kerberos ticket cache

PyArrow’s libhdfs driver can also be affected by a few environment variables. For more information on these, see the pyarrow documentation.

4.35.4 Amazon S3

Amazon S3 (Simple Storage Service) is a web service offered by Amazon Web Services.

The S3 back-end available to Dask is s3fs, and 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 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* or *endpoint_url*. Notice: do not pass the *config* option here, please pass its content to *config_kwargs* instead.
- *config_kwargs*: Dict passed to the s3fs.S3FileSystem, which passes it to the boto3 client’s *config* option.
- *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)
Using Other S3-Compatible Services

By using the `endpoint_url` option, you may use other s3-compatible services, for example, using AlibabaCloud OSS:

```python
dask_function(...,
    storage_options={
        "key": ..., 
        "secret": ..., 
        "client_kwargs": {
            "endpoint_url": "http://some-region.some-s3-compatible.com",
        },
        # this dict goes to boto3 client's `config`
        # `addressing_style` is required by AlibabaCloud, other services may not
        "config_kwargs": {
            "s3": {
                "addressing_style": "virtual"},
        }
    })
```

4.35.5 Google Cloud Storage

Google Cloud Storage is a RESTful online file storage web service for storing and accessing data on Google’s infrastructure.

The GCS back-end is identified by the protocol identifiers `gcs` and `gs`, which are identical in their effect.

Multiple modes of authentication are supported. These options should be included in the `storage_options` dictionary as `{'token': ..}` submitted with your call to a storage-based Dask function/method. See the `gcsfs` documentation for further details.

General recommendations for distributed clusters, in order:

- use ‘anon’ for public data
- use ‘cloud’ if this is available
- use `gcloud` to generate a JSON file, and distribute this to all workers, and supply the path to the file
- use `gcsfs` directly with the ‘browser’ method to generate a token cache file (`~/.gcs_tokens`) and distribute this with method ‘cache’

The final suggestion may be the fastest and simplest for authenticated access (as opposed to anonymous), since it will not require re-authentication. However, this method is not secure since credentials will be passed directly around the cluster. This is fine if you are certain that the cluster is itself secured. You need to create a `GCSFileSystem` object using any method that works for you and then pass its credentials directly:

```python
gcs = GCSFileSystem(...) 
dask_function(..., storage_options={'token': gcs.session.credentials})
```

4.35.6 Azure Datalake

**Warning:** Support for ADL is experimental - use at own risk.

Parameters `tenant_id`, `client_id`, and `client_secret` are required for authentication in `storage_options`, and all other parameters will be passed on to the `AzureDLFileSystem` constructor (follow the link for further information). The auth parameters are passed directly to workers, so this should only be used within a secure cluster.
4.35.7 HTTP(s)

Direct file-like access to arbitrary URLs is available over HTTP and HTTPS. However, there is no such thing as `glob` functionality over HTTP, so only explicit lists of files can be used.

Server implementations differ in the information they provide - they may or may not specify the size of a file via a HEAD request or at the start of a download - and some servers may not respect byte range requests. The HTTP-FileSystem therefore offers best-effort behaviour: the download is streamed but, if more data is seen than the configured block-size, an error will be raised. To be able to access such data you must read the whole file in one shot (and it must fit in memory).

Note that, currently, http:// and https:// are treated as separate protocols, and cannot be mixed.

4.35.8 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:

```python
from dask import bytes

dask.bytes.core._filesystems['myproto'] = MyProtoFileSystem
```

For a more complicated example, users may wish to also see `dask.bytes.s3.DaskS3FileSystem`.

```python
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

    mkdirs(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.
        kwrgs: 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
```

4.36 GPUs

Dask works with GPUs in a few ways.
4.36.1 Custom Computations

Many people use Dask alongside GPU-accelerated libraries like PyTorch and TensorFlow to manage workloads across several machines. They typically use Dask’s custom APIs, notably `Delayed` and `Futures`.

Dask doesn’t need to know that these functions use GPUs. It just runs Python functions. Whether or not those Python functions use a GPU is orthogonal to Dask. It will work regardless.

As a worked example, you may want to view this talk:

4.36.2 High Level Collections

Dask can also help to scale out large array and dataframe computations by combining the Dask Array and DataFrame collections with a GPU-accelerated array or dataframe library.

Recall that `Dask Array` creates a large array out of many NumPy arrays and `Dask DataFrame` creates a large dataframe out of many Pandas dataframes. We can use these same systems with GPUs if we swap out the NumPy/Pandas components with GPU-accelerated versions of those same libraries, as long as the GPU accelerated version looks enough like NumPy/Pandas in order to interoperate with Dask.

Fortunately, libraries that mimic NumPy, Pandas, and Scikit-Learn on the GPU do exist.

**DataFrames**

The RAPIDS libraries provide a GPU accelerated Pandas-like library, `cuDF`, which interoperates well and is tested against Dask DataFrame.

If you have cudf installed then you should be able to convert a Pandas-backed Dask DataFrame to a cuDF-backed Dask DataFrame as follows:

```python
import cudf

df = df.map_partitions(cudf.from_pandas)  # convert pandas partitions into cudf
```

However, cuDF does not support the entire Pandas interface, and so a variety of Dask DataFrame operations will not function properly. Check the [cuDF API Reference](https://docs.rapids.ai/api/cudf_reference) for currently supported interface.

**Arrays**

**Note:** Dask’s integration with CuPy relies on features recently added to NumPy and CuPy, particularly in version `numpy>=1.17` and `cupy>=6`

Chainer’s `CuPy` library provides a GPU accelerated NumPy-like library that interoperates nicely with Dask Array.

If you have CuPy installed then you should be able to convert a NumPy-backed Dask Array into a CuPy backed Dask Array as follows:

```python
import cupy

x = x.map_blocks(cupy.asarray)
```

CuPy is fairly mature and adheres closely to the NumPy API. However, small differences do exist and these can cause Dask Array operations to function improperly. Check the [CuPy Reference Manual](https://cupy.readthedocs.io/en/stable/) for API compatibility.
Scikit-Learn

There are a variety of GPU accelerated machine learning libraries that follow the Scikit-Learn Estimator API of fit, transform, and predict. These can generally be used within Dask-ML’s meta estimators, such as hyper parameter optimization.

Some of these include:

- Skorch
- cuML
- LightGBM
- XGBoost
- Thunder SVM
- Thunder GBM

4.36.3 Setup

From the examples above we can see that the user experience of using Dask with GPU-backed libraries isn’t very different from using it with CPU-backed libraries. However, there are some changes you might consider making when setting up your cluster.

Restricting Work

By default Dask allows as many tasks as you have CPU cores to run concurrently. However if your tasks primarily use a GPU then you probably want far fewer tasks running at once. There are a few ways to limit parallelism here:

- Limit the number of threads explicitly on your workers using the \texttt{--nthreads} keyword in the CLI or the \texttt{ncores=} keyword the Cluster constructor.
- Use worker resources and tag certain tasks as GPU tasks so that the scheduler will limit them, while leaving the rest of your CPU cores for other work

Specifying GPUs per Machine

Some configurations may have many GPU devices per node. Dask is often used to balance and coordinate work between these devices.

In these situations it is common to start one Dask worker per device, and use the CUDA environment variable \texttt{CUDA_VISIBLE_DEVICES} to pin each worker to prefer one device.

\begin{verbatim}
# If we have four GPUs on one machine
CUDA_VISIBLE_DEVICES=0 dask-worker ...
CUDA_VISIBLE_DEVICES=1 dask-worker ...
CUDA_VISIBLE_DEVICES=2 dask-worker ...
CUDA_VISIBLE_DEVICES=3 dask-worker ...
\end{verbatim}

The Dask CUDA project contains some convenience CLI and Python utilities to automate this process.

4.36.4 Work in Progress

GPU computing is a quickly moving field today and as a result the information in this page is likely to go out of date quickly. We encourage interested readers to check out Dask’s Blog which has more timely updates on ongoing work.
4.37 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 https://dask.org

A BibTeX entry for LaTeX users follows:

```latex
@Manual{,
  title = {Dask: Library for dynamic task scheduling},
  author = {{Dask Development Team}},
  year = {2016},
  url = {https://dask.org},
}
```

The full author list is available using git (e.g. `git shortlog -ns`).

4.37.1 Papers about parts of Dask


```latex
@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 }
}
```

4.38 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.
4.39 Images and Logos

Dask is supported by Anaconda Inc and develops under the BSD 3-clause license.


Symbols

-bokeh, -no-bokeh
  dask-scheduler command line option, 17
  dask-worker command line option, 18
-bokeh-port <bokeh_port>
  dask-scheduler command line option, 17
  dask-worker command line option, 18
-bokeh-prefix <bokeh_prefix>
  dask-scheduler command line option, 17
  dask-worker command line option, 19
-bokeh-whitelist <bokeh_whitelist>
  dask-scheduler command line option, 17
  dask-worker command line option, 18
-contact-address <contact_address>
  dask-worker command line option, 19
-dashboard-address <dashboard_address>
  dask-scheduler command line option, 17
  dask-worker command line option, 18
-death-timeout <death_timeout>
  dask-worker command line option, 18
-host <host>
  dask-remote command line option, 21
  dask-scheduler command line option, 17
-hostfile <hostfile>
  dask-scheduler command line option, 20
-interface <interface>
  dask-scheduler command line option, 17
  dask-worker command line option, 19
-listen-address <listen_address>
  dask-worker command line option, 18
-local-directory <local_directory>
  dask-scheduler command line option, 18
-log-directory <log_directory>
  dask-ssh command line option, 20
-memory-limit <memory_limit>
  dask-scheduler command line option, 20
  dask-worker command line option, 19
-name <name>
  dask-worker command line option, 19
-nanny, -no-nanny
  dask-worker command line option, 19
-nanny-port <nanny_port>
  dask-scheduler command line option, 20
  dask-worker command line option, 18
-nohost
  dask-ssh command line option, 20
-nprocs <nprocs>
  dask-scheduler command line option, 20
  dask-worker command line option, 19
-ntthreads <ntthreads>
  dask-scheduler command line option, 20
  dask-worker command line option, 19
-pid-file <pid_file>
  dask-scheduler command line option, 18
  dask-worker command line option, 19
-port <port>
  dask-remote command line option, 21
  dask-scheduler command line option, 17
-preload <preload>
  dask-scheduler command line option, 18
  dask-worker command line option, 19
-reconnect, -no-reconnect
  dask-worker command line option, 19
-remote-dask-worker
  <remote_dask_worker>
  dask-scheduler command line option, 20
-remote-python <remote_python>
dask-ssh command line option, 20
resources <resources>
dask-worker command line option, 19
-scheduler <scheduler>
dask-ssh command line option, 20
-scheduler-file <scheduler_file>
dask-scheduler command line option, 18
dask-worker command line option, 19
-scheduler-port <scheduler_port>
dask-ssh command line option, 20
-show, -no-show
dask-scheduler command line option, 17
-ssh-port <ssh_port>
dask-ssh command line option, 20
-ssh-private-key <ssh_private_key>
dask-ssh command line option, 20
-ssh-username <ssh_username>
dask-ssh command line option, 20
tls-ca-file <tls_ca_file>
dask-scheduler command line option, 17
dask-worker command line option, 18
tls-cert <tls_cert>
dask-scheduler command line option, 17
dask-worker command line option, 18
-use-xheaders <use_xheaders>
dask-scheduler command line option, 18
-worker-port <worker_port>
dask-scheduler command line option, 20
dask-worker command line option, 18
__dask_graph___(), 829
__dask_keys___(), 830
__dask_optimize__(), 830
__dask_postcompute__(), 830
__dask_postpersist__(), 830
__dask_scheduler__(), 830
add_done_callback() (distributed.Future method), 783
agg() (dask.dataframe.groupby.DataFrameGroupBy method), 680
agg() (dask.dataframe.groupby.SeriesGroupBy method), 687
aggregate() (dask.dataframe.groupby.DataFrameGroupBy method), 682
aggregate() (dask.dataframe.groupby.SeriesGroupBy method), 688
Aggregation (class in dask.dataframe), 694
align() (dask.dataframe.DataFrame method), 405
align() (dask.dataframe.Series method), 573
all() (dask.array.Array method), 328
all() (dask.dataframe.DataFrame method), 376
all() (dask.dataframe.Series method), 574
all() (module dask.array), 66
allclose() (in module dask.array), 67
angle() (in module dask.array), 68
any() (dask.array.Array method), 329
any() (dask.array.Array method), 376
any() (dask.dataframe.DataFrame method), 408
any() (dask.dataframe.Series method), 576
any() (module dask.array), 69
append() (dask.dataframe.DataFrame method), 410
append() (dask.dataframe.Series method), 578
apply() (dask.dataframe.DataFrame method), 411
apply() (dask.dataframe.groupby.DataFrameGroupBy method), 683
apply() (dask.dataframe.groupby.SeriesGroupBy method), 690
apply() (dask.dataframe.Series method), 579
apply_along_axis() (in module dask.array), 70
apply_gufunc() (in module dask.array.gufunc), 319
apply_over_axes() (in module dask.array), 71
applymap() (dask.dataframe.DataFrame method), 413
arg_all() (in module dask.array), 72
arccos() (in module dask.array), 73
arccosh() (in module dask.array), 74
arcsin() (in module dask.array), 74
arcsinh() (in module dask.array), 75
arctan() (in module dask.array), 76
arctan2() (in module dask.array), 77
arctanh() (in module dask.array), 79
argmax() (dask.array.Array method), 329
argmax() (in module dask.array), 79
argmin() (dask.array.Array method), 329
argmin() (in module dask.array), 80
argtopk() (dask.array.Array method), 329
argtopk() (in module dask.array), 81
argwhere() (in module dask.array), 82
around() (in module dask.array), 83
Array (class in dask.array), 328

A
add() (dask.dataframe.DataFrame method), 402
add() (dask.dataframe.Series method), 571
accumulate() (dask.bag.Bag method), 375
acquire() (distributed.Lock method), 786
Adaptive (class in distributed.deploy), 34
add() (dask.dataframe.DataFrame method), 403
add() (dask.dataframe.Series method), 573

922  Index
array() (in module dask.array), 83
as_completed() (in module distributed), 760
as_gufunc() (in module dask.array.gufunc), 320
asanyarray() (in module dask.array), 85
asarray() (in module dask.array), 86
assign() (dask.dataframe.DataFrame method), 413
astype() (dask.dataframe.Array method), 329
astype() (dask.dataframe.DataFrame method), 415
astype() (dask.dataframe.Series method), 580
asynchronous (distributed.Client attribute), 764
atleast_1d() (in module dask.array), 86
atleast_2d() (in module dask.array), 87
atleast_3d() (in module dask.array), 87
autocorr() (dask.dataframe.Series method), 581
average() (in module dask.array), 88
average() (in module dask.array.ma), 233

B
Bag (class in dask.bag), 375
beta() (in module dask.array.random), 272
between() (dask.dataframe.Series method), 582
bfill() (dask.dataframe.DataFrame method), 416
bfill() (dask.dataframe.Series method), 583
bincount() (in module dask.array), 89
binomial() (in module dask.array.random), 272
bitwise_and() (in module dask.array), 90
bitwise_not() (in module dask.array), 91
bitwise_or() (in module dask.array), 93
bitwise_xor() (in module dask.array), 94
block() (in module dask.array), 95
blocks (dask.array.Array attribute), 330
blockwise() (in module dask.array.core), 325
broadcast_arrays() (in module dask.array), 96
broadcast_to() (in module dask.array), 96

C
CacheProfiler() (in module dask.diagnostics), 806
call_stack() (distributed.Client method), 764
Callback() (in module dask.diagnostics), 807
cancel() (distributed.Client method), 765
cancel() (distributed.Future method), 784
cancelled() (distributed.Future method), 784
categorize() (dask.dataframe.DataFrame method), 416
ceil() (in module dask.array), 97
chisquare() (in module dask.array.random), 273
chisquare() (in module dask.array.stats), 309
choice() (in module dask.array.random), 274
cholesky() (in module dask.array.linalg), 227
choose() (dask.array.Array method), 330
choose() (in module dask.array), 98
clear_divisions() (dask.dataframe.DataFrame method), 416
clear_divisions() (dask.dataframe.Series method), 583
Client (class in distributed), 763
clip() (dask.array.Array method), 330
clip() (dask.dataframe.DataFrame method), 416
clip() (dask.dataframe.Series method), 583
clip() (in module dask.array), 100
clip_lower() (dask.dataframe.DataFrame method), 418
clip_lower() (dask.dataframe.Series method), 585
clip_upper() (dask.dataframe.DataFrame method), 419
clip_upper() (dask.dataframe.Series method), 586
close() (distributed.Client method), 765
close() (distributed.deploy.local.LocalCluster method), 15
Cluster (class in distributed.deploy), 35
coarsen() (in module dask.array), 64, 97
collect() (in module dask.config), 895
combine() (dask.dataframe.DataFrame method), 421
combine() (dask.dataframe.Series method), 588
combine_first() (dask.dataframe.DataFrame method), 422
combine_first() (dask.dataframe.Series method), 589
compress() (in module dask.array), 101
compute() (dask.dataframe.DataFrame method), 423
compute() (dask.dataframe.Series method), 589
compute() (distributed.Client method), 765
compute() (in module dask), 792
compute() (in module dask dataframe), 711
concat() (in module dask.bag), 390
concat() (in module dask.dataframe.multi), 712
concatenate() (in module dask.array), 65, 102
copy() (in module dask.array), 102
copy() (dask.dataframe.DataFrame method), 423
copy() (dask.dataframe.Series method), 589
copysign() (in module dask.array), 103
corr() (dask.dataframe.DataFrame method), 424
corr() (dask.dataframe.Series method), 590
corrcoef() (in module dask.array), 103
cos() (in module dask.array), 104
cosh() (in module dask.array), 105
count() (dask.bag.Bag method), 376
count() (dask.dataframe.DataFrame method), 424
count() (dask.dataframe.groupby.DataFrameGroupBy method), 684
count() (dask.dataframe.groupby.SeriesGroupBy method), 690
count() (dask.dataframe.Series method), 590
count_nonzero() (in module dask.array), 106
cov() (dask.dataframe.DataFrame method), 426
cov() (dask.dataframe.Series method), 590
cov() (in module dask.array), 106

cull() (in module dask.optimization), 825

cumcount() (dask.dataframe.groupby.DataFrameGroupBy method), 684
cumcount() (dask.dataframe.groupby.SeriesGroupBy method), 690
cummax() (dask.dataframe.DataFrame method), 427
cummax() (dask.dataframe.Series method), 592
cummin() (dask.dataframe.DataFrame method), 429
cummin() (dask.dataframe.Series method), 594
cumprod() (dask.array.Array method), 331
cumprod() (dask.dataframe.DataFrame method), 430
cumprod() (dask.dataframe.groupby.DataFrameGroupBy method), 685
cumprod() (dask.dataframe.groupby.SeriesGroupBy method), 691
cumprod() (dask.dataframe.Series method), 596
cumsum() (dask.array.Array method), 331
cumsum() (dask.dataframe.DataFrame method), 432
cumsum() (dask.dataframe.groupby.DataFrameGroupBy method), 685
cumsum() (dask.dataframe.groupby.SeriesGroupBy method), 691
cumsum() (dask.dataframe.Series method), 598
cumsum() (in module dask.array), 108
current() (distributed.Client class method), 766

dask-remote command line option
 -host <host>, 21
-port <port>, 21
dask-scheduler command line option
 -bokeh, -no-bokeh, 17
-bokeh-port <bokeh_port>, 17
-bokeh-prefix <bokeh_prefix>, 17
-bokeh-whitelist <bokeh_whitelist>, 17
-dashboard-address <dashboard_address>, 17
-host <host>, 17
-interface <interface>, 17
-local-directory <local_directory>, 18
-pid-file <pid_file>, 18
-port <port>, 17
-preload <preload>, 18
-scheduler-file <scheduler_file>, 18
-show, -no-show, 17
-tls-ca-file <tls_ca_file>, 17
tls-cert <tls_cert>, 17
tls-key <tls_key>, 17
-use-xheaders <use_xheaders>, 18
PRELOAD_ARGV, 18

dask-ssh command line option
 -hostfile <hostfile>, 20
-log-directory <log_directory>, 20
-memory-limit <memory_limit>, 20
-nanny-port <nanny_port>, 20
-nhost, 20
-nprocs <nprocs>, 20
-nthreads <nthreads>, 20
-remote-dask-worker <remote_dask_worker>, 20
-remote-python <remote_python>, 20
-scheduler <scheduler>, 20
-scheduler-port <scheduler_port>, 20
-ssh-port <ssh_port>, 20
-ssh-private-key <ssh_private_key>, 20
-ssh-username <ssh_username>, 20
-worker-port <worker_port>, 20

PRELOAD_ARGV, 18

PRELOAD_ARGV, 18

SCHEDULER, 19

dask-submit command line option
 FILEPATH, 21
REMOTE_CLIENT_ADDRESS, 21

dask-worker command line option
-bokeh, -no-bokeh, 18
-bokeh-port <bokeh_port>, 18
-bokeh-prefix <bokeh_prefix>, 19
-contact-address <contact_address>, 18
-dashboard-address <dashboard_address>, 18
-death-timeout <death_timeout>, 19
-host <host>, 19
-interface <interface>, 19
-local-directory <local_directory>, 19
-memory-limit <memory_limit>, 19
-name <name>, 19
-nanny, -no-nanny, 19
-nanny-port <nanny_port>, 18
-nprocs <nprocs>, 19
-nthreads <nthreads>, 19
-pid-file <pid_file>, 19
-preload <preload>, 19
-reconnect, -no-reconnect, 19
-resources <resources>, 19
-scheduler-file <scheduler_file>, 19
-tls-ca-file <tls_ca_file>, 19
tls-cert <tls_cert>, 19
tls-key <tls_key>, 19
-worker-port <worker_port>, 19
PRELOAD_ARGV, 19
SCHEDULER, 19
max() (dask.array.Array method), 335
max() (dask.bag.Bag method), 381
max() (dask.dataframe.DataFrame method), 483
max() (dask.dataframe.groupby.DataFrameGroupBy method), 686
max() (dask.dataframe.groupby.SeriesGroupBy method), 692
max() (dask.dataframe.Series method), 627
max() (in module dask.array), 163
maximum() (in module dask.array), 164
mean() (dask.array.Array method), 335
mean() (dask.bag.Bag method), 381
mean() (dask.dataframe.DataFrame method), 484
mean() (dask.dataframe.groupby.DataFrameGroupBy method), 686
mean() (dask.dataframe.groupby.SeriesGroupBy method), 692
mean() (dask.dataframe.Series method), 628
mean() (in module dask.array), 165
melt() (in module dask.dataframe.reshape), 717
memory_usage() (dask.dataframe.DataFrame method), 485
memory_usage() (dask.dataframe.Series method), 628
merge() (dask.dataframe.DataFrame method), 486
merge() (in module dask.config), 894
merge() (in module dask.dataframe.multi), 713
meshgrid() (in module dask.array), 167
min() (dask.array.Array method), 335
min() (dask.bag.Bag method), 381
min() (dask.dataframe.DataFrame method), 487
min() (dask.dataframe.groupby.DataFrameGroupBy method), 686
min() (dask.dataframe.groupby.SeriesGroupBy method), 693
min() (dask.dataframe.Series method), 629
min() (in module dask.array), 168
minimum() (in module dask.array), 170
mkdirs() (dask.bytes.local.LocalFileSystem method), 909
mod() (dask.dataframe.DataFrame method), 489
mod() (dask.dataframe.Series method), 631
modf() (in module dask.array), 171
moment() (dask.array.Array method), 335
moment() (in module dask.array), 171
moment() (in module dask.array.stats), 317
mul() (dask.dataframe.DataFrame method), 491
mul() (dask.dataframe.Series method), 631

N
nan_to_num() (in module dask.array), 182
nanargmax() (in module dask.array), 172
nanargmin() (in module dask.array), 172
nancumprod() (in module dask.array), 172
nancumsum() (in module dask.array), 172
nanmax() (in module dask.array), 173
nanmean() (in module dask.array), 175
nanmin() (in module dask.array), 176
nanprod() (in module dask.array), 177
nanstd() (in module dask.array), 178
nansum() (in module dask.array), 179
nanvar() (in module dask.array), 180
nbytes (dask.array.Array attribute), 336
nbytes (dask.dataframe.Series attribute), 632
nbytes() (distributed.Client method), 771
ncores() (distributed.Client method), 771
ndim (dask.dataframe.DataFrame attribute), 494
ndim (dask.dataframe.Series attribute), 632
ne() (dask.dataframe.DataFrame method), 494
ne() (dask.dataframe.Series method), 632
negative_binomial() (in module dask.array.random), 286
next() (distributed.Sub method), 788
nextafter() (in module dask.array), 183
nlargest() (dask.dataframe.DataFrame method), 497
nlargest() (dask.dataframe.Series method), 633
noncentral_chisquare() (in module dask.array.random), 287
noncentral_f() (in module dask.array.random), 288
nonzero() (dask.array.Array method), 336
nonzero() (in module dask.array), 183
norm() (in module dask.array.linalg), 228
normal() (in module dask.array.random), 289
normalize_chunks() (in module dask.array.core), 327
normalize_collection() (distributed.Client method), 771
notnulltest() (in module dask.array.stats), 315
notnull() (dask.dataframe.DataFrame method), 499
notnull() (dask.dataframe.Series method), 635
notnull() (in module dask.array), 185
npartitions (dask.dataframe.DataFrame attribute), 500
npartitions (dask.dataframe.Series attribute), 636
nsmmallest() (dask.dataframe.DataFrame method), 500
nsmmallest() (dask.dataframe.Series method), 636
nunique() (dask.dataframe.Series method), 638
nunique_approx() (dask.dataframe.DataFrame method), 501
nunique_approx() (dask.dataframe.Series method), 638

O
ones() (in module dask.array), 185
ones_like() (in module dask.array), 186
open() (dask.bytes.local.LocalFileSystem method), 909
open_files() (in module dask.bytes), 904
optimize() (in module dask), 793
outer() (in module dask.array), 186
overlap() (in module dask.array.overlap), 245

P
pad() (in module dask.array), 187
pareto() (in module dask.array.random), 290
partitions (dask.dataframe.DataFrame attribute), 502
partitions (dask.dataframe.Series attribute), 638
percentile() (in module dask.array), 190
persist() (dask.dataframe.DataFrame method), 502
persist() (dask.dataframe.Series method), 638
persist() (distributed.Client method), 772
persist() (in module dask), 793
piecewise() (in module dask.array), 191
pipe() (dask.dataframe.DataFrame method), 503
pipe() (dask.dataframe.Series method), 639
pivot_table() (dask.dataframe.DataFrame method), 503
pivot_table() (in module dask.dataframe.reshape), 717
pluck() (dask.bag.Bag method), 381
poisson() (in module dask.array.random), 291
pow() (dask.dataframe.DataFrame method), 504
pow() (dask.dataframe.Series method), 640
power() (in module dask.array.random), 292
power_divergence() (in module dask.array.stats), 310
PRELOAD_ARGV
dask-scheduler command line option, 18
dask-worker command line option, 19
processing() (distributed.Client method), 772
prod() (dask.array.Array method), 336
prod() (dask.dataframe.DataFrame method), 506
prod() (dask.dataframe.groupby.DataFrameGroupBy method), 686
prod() (dask.dataframe.groupby.SeriesGroupBy method), 693
prod() (dask.dataframe.Series method), 641
prod() (in module dask.array), 192
product() (dask.bag.Bag method), 382
profile() (distributed.Client method), 773
Profiler() (in module dask.diagnostics), 804
progress() (in module dask.distributed), 809
ProgressBar() (in module dask.diagnostics), 804
ptp() (in module dask.array), 194
Pub (class in distributed), 756, 787
publish_dataset() (distributed.Client method), 773
put() (distributed.Pub method), 757, 788
put() (distributed.Queue method), 785
Q
qr() (in module dask.array.linalg), 231
qsize() (distributed.Queue method), 785
quantile() (dask.dataframe.DataFrame method), 507
quantile() (dask.dataframe.Series method), 642
query() (dask.dataframe.DataFrame method), 507
Queue (class in distributed), 784
R
rad2deg() (in module dask.array), 194
radd() (dask.dataframe.DataFrame method), 508
radd() (dask.dataframe.Series method), 642
radiants() (in module dask.array), 195
randint() (in module dask.array.random), 293
random() (in module dask.array.random), 294
random_sample() (dask.bag.Bag method), 382
random_sample() (in module dask.array.random), 295
random_split() (dask.dataframe.DataFrame method), 510
random_split() (dask.dataframe.Series method), 643
range() (in module dask.array), 390
ravel() (dask.array.Array method), 336
ravel() (in module dask.array), 196
rayleigh() (in module dask.array.random), 295
rdiv() (dask.dataframe.DataFrame method), 511
rdiv() (dask.dataframe.Series method), 643
read_avro() (in module dask.array), 390
read_bytes() (in module dask.array), 903
read_csv() (in module dask.array), 695
read_fwf() (in module dask.array), 697
read_hdf() (in module dask.array), 699
read_json() (in module dask.array), 700
read_orc() (in module dask.array), 699
read_parquet() (in module dask.array), 698
read_sql_table() (in module dask.array), 701
read_table() (in module dask.array), 696
read_text() (in module dask.array), 388
real() (in module dask.array), 197
rebalance() (distributed.Client method), 774
rechunk() (dask.array.Array method), 337
rechunk() (in module dask.array), 198
reduction() (dask.bag.Bag method), 382
reduction() (dask.dataframe.DataFrame method), 514
reduction() (dask.dataframe.Series method), 644
refresh() (in module dask.config), 895
register_worker_callbacks() (distributed.Client method), 774
rejoin() (in module distributed), 762
release() (distributed.Lock method), 787
REMOTE_CLIENT_ADDRESS
dask-submit command line option, 21
remove() (dask.bag.Bag method), 382
rename() (dask.dataframe.DataFrame method), 515
rename() (dask.dataframe.Series method), 646
repartition() (dask.bag.Bag method), 383
repartition() (dask.dataframe.DataFrame method), 517
repartition() (dask.dataframe.Series method), 646
repeat() (dask.array.Array method), 337
repeat() (in module dask.array), 199
replace() (dask.dataframe.DataFrame method), 517
replace() (dask.dataframe.Series method), 647
replicate() (dask.distributed.Client method), 774
resample() (dask.dataframe.DataFrame method), 522
resample() (dask.dataframe.Series method), 651
reset_index() (dask.dataframe.DataFrame method), 527
reset_index() (dask.dataframe.Series method), 656
reshape() (dask.array.Array method), 337
reshape() (in module dask.array), 199
ResourceProfiler() (in module dask.diagnostics), 805
restart() (distributed.Client method), 775
result() (distributed.Future method), 784
result_type() (in module dask.array), 199
retire_workers() (dask.distributed.Client method), 775
retry() (dask.distributed.Client method), 775
retry() (dask.distributed.Future method), 784
RewriteRule() (in module dask.rewrite), 827
rfft() (in module dask.array.fft), 261
rfftfreq() (in module dask.array.fft), 262
rfftn() (in module dask.array.fft), 263
rfloordiv() (dask.dataframe.DataFrame method), 527
rfloordiv() (dask.dataframe.Series method), 656
rint() (in module dask.array), 200
rmod() (dask.dataframe.DataFrame method), 530
rmod() (dask.dataframe.Series method), 657
rmul() (dask.dataframe.DataFrame method), 532
rmul() (dask.dataframe.Series method), 658
roll() (in module dask.array), 201
rolling() (dask.dataframe.DataFrame method), 535
rolling() (dask.dataframe.Series method), 659
round() (dask.array.Array method), 337
round() (dask.dataframe.DataFrame method), 536
round() (dask.dataframe.Series method), 659
round() (in module dask.array), 202
rpow() (dask.dataframe.DataFrame method), 537
rpow() (dask.dataframe.Series method), 660
rsub() (dask.dataframe.DataFrame method), 539
rsub() (dask.dataframe.Series method), 661
rtruediv() (dask.dataframe.DataFrame method), 542
rtruediv() (dask.dataframe.Series method), 662
RuleSet() (in module dask.rewrite), 828
run() (distributed.Client method), 775
run_coroutine() (distributed.Client method), 776
run_on_scheduler() (distributed.Client method), 777
S
sample() (dask.dataframe.DataFrame method), 545
sample() (dask.dataframe.Series method), 662
scale_down() (distributed.deploy.local.LocalCluster method), 15
scale_up() (distributed.deploy.local.LocalCluster method), 15
scatter() (distributed.Client method), 777
SCHEDULER
dask-worker command line option, 19
scheduler_info() (distributed.Client method), 778
secede() (in module distributed), 762
select_dtypes() (dask.dataframe.DataFrame method), 545
sem() (dask.dataframe.DataFrame method), 547
sem() (dask.dataframe.Series method), 663
Series (class in dask.dataframe), 571
SeriesGroupBy (class in dask.dataframe.groupby), 687
set() (distributed.Variable method), 786
set() (in module dask.config), 894
set_fill_value() (in module dask.array.ma), 244
set_index() (dask.dataframe.DataFrame method), 547
set_metadata() (distributed.Client method), 779
sfqr() (in module dask.array.linalg), 232
shape (dask.dataframe.DataFrame attribute), 548
shape (dask.dataframe.Series attribute), 663
shift() (dask.dataframe.DataFrame method), 548
shift() (dask.dataframe.Series method), 663
shutdown() (distributed.Client method), 779
sign() (in module dask.array), 202
signbit() (in module dask.array), 202
sin() (in module dask.array), 203
sinh() (in module dask.array), 204
size (dask.array.Array attribute), 337
size (dask.dataframe.DataFrame attribute), 550
size (dask.dataframe.Series attribute), 665
size() (dask.bytes.local.LocalFileSystem method), 909
size() (dask.dataframe.groupby.DataFrameGroupBy method), 686
size() (dask.dataframe.groupby.SeriesGroupBy method), 693
skew() (in module dask.array.stats), 313
skewtest() (in module dask.array.stats), 313
solve() (in module dask.array.linalg), 231
solve_triangular() (in module dask.array.linalg), 231
sqrt() (in module dask.array), 205
squeeze () (in module dask.array), 206
squeeze () (dask.array.Array method), 337
squeeze () (dask.dataframe.DataFrame method), 550
squeeze () (dask.dataframe.Series method), 665
squeeze () (in module dask.array), 206
stack () (in module dask.array), 65, 207
standard_cauchy () (in module
dask.array.random), 296
standard_exponential () (in module
dask.array.random), 297
standard_gamma () (in module dask.array.random), 298
standard_normal () (in module
dask.array.random), 299
standard_t () (in module dask.array.random), 299
starmap () (dask.array.Bag method), 383
start () (distributed.Client method), 779
start_ipython_scheduler () (distributed.Client method), 779
start_ipython_workers () (distributed.Client method), 780
start_worker () (distributed.deploy.local.LocalCluster method), 15
std () (dask.array.Array method), 338
std () (dask.array.Bag method), 383
std () (dask.dataframe.DataFrame method), 551
std () (dask.dataframe.groupby.DataFrameGroupBy method), 687
std () (dask.dataframe.groupby.SeriesGroupBy method), 693
std () (dask.dataframe.Series method), 666
std () (in module dask.array), 207
stop_worker () (distributed.deploy.local.LocalCluster method), 15
store () (dask.array.Array method), 338
store () (in module dask.array), 63, 249
str (dask.array.Bag attribute), 384
str (dask.dataframe.Series attribute), 667
Sub (class in distributed), 788
sub () (dask.dataframe.DataFrame method), 552
sub () (dask.dataframe.Series method), 667
submit () (distributed.Client method), 781
sum () (dask.array.Array method), 339
sum () (dask.array.Bag method), 384
sum () (dask.dataframe.DataFrame method), 555
sum () (dask.dataframe.groupby.DataFrameGroupBy method), 687
sum () (dask.dataframe.groupby.SeriesGroupBy method), 693
sum () (dask.dataframe.Series method), 668
sum () (in module dask.array), 209
svd () (in module dask.array.linalg), 231
svd_compressed () (in module dask.array.linalg), 232
swapaxes () (dask.array.Array method), 339
T
tail () (dask.dataframe.DataFrame method), 556
tail () (dask.dataframe.Series method), 670
take () (dask.array.Bag method), 384
take () (in module dask.array), 210
tan () (in module dask.array), 212	
tanh () (in module dask.array), 213
tensordot () (in module dask.array), 214
tile () (in module dask.array), 216
to_avro () (dask.array.Bag method), 384
to_avro () (in module dask.array.avro), 371
to_bag () (dask.dataframe.DataFrame method), 556
to_bag () (dask.dataframe.Series method), 670
to_bag () (in module dask.array), 707
to_csv () (dask.dataframe.DataFrame method), 557
to_csv () (dask.dataframe.Series method), 670
to_csv () (in module dask.array), 705
to_dask_array () (dask.dataframe.DataFrame method), 558
to_dask_array () (dask.dataframe.Series method), 671
to_dask_dataframe () (dask.array.Array method), 339
to_dask_dataframe () (dask.array.Bag method), 385
to_dask_dataframe () (dask.array.core.Bag method), 372
to_delayed () (dask.array.Array method), 339
to_delayed () (dask.array.Bag method), 386
to_delayed () (dask.array.core.Bag method), 373
to_delayed () (dask.dataframe.DataFrame method), 559
to_delayed () (dask.dataframe.Series method), 672
to_frame () (dask.dataframe.Series method), 672
to_hdf () (dask.dataframe.DataFrame method), 559
to_hdf () (dask.dataframe.Series method), 672
to_hdf () (in module dask.array), 708
to_hdf5 () (dask.array.Array method), 340
to_hdf5 () (in module dask.array), 250
to_html () (dask.dataframe.DataFrame method), 560
to_json () (dask.dataframe.DataFrame method), 562
to_json () (dask.dataframe.Series method), 674
to_json () (in module dask.array), 710
to_numpy_stack () (in module dask.array), 251
to_parquet () (dask.dataframe.DataFrame method), 562
to_parquet () (in module dask.array), 709
to_records () (dask.dataframe.DataFrame method), 563
to_records () (in module dask.array), 705
to_string () (dask.dataframe.DataFrame method), 563
to_string() (dask.dataframe.Series method), 674
to_textfiles() (dask.bag.Bag method), 386
to_textfiles() (in module dask.bag.core), 370
to_tiledb() (dask.array.Array method), 340
to_tiledb() (in module dask.array), 252
to_timestamp() (dask.dataframe.DataFrame method), 565
to_timestamp() (dask.dataframe.Series method), 674
to_zarr() (dask.array.Array method), 340
to_zarr() (in module dask.array), 251
topk() (dask.array.Array method), 340
topk() (dask.bag.Bag method), 387
topk() (in module dask.array), 217
trace() (dask.array.Array method), 340
traceback() (distributed.Future method), 784
transpose() (dask.array.Array method), 340
transpose() (in module dask.array), 217
triangular() (in module dask.array.random), 300
tril() (in module dask.array), 218
trim_internal() (in module dask.array.overlap), 247
trim_overlap() (in module dask.array.overlap), 247
triu() (in module dask.array), 218
truediv() (dask.dataframe.DataFrame method), 565
truediv() (dask.dataframe.Series method), 674
trunc() (in module dask.array), 219
tsqr() (in module dask.array.linalg), 233
ttest_1samp() (in module dask.array.stats), 307
ttest_ind() (in module dask.array.stats), 305
ttest_rel() (in module dask.array.stats), 308

U
ukey() (dask.bytes.local.LocalFileSystem method), 909
uniform() (in module dask.array.random), 301
unique() (dask.dataframe.Series method), 675
unique() (in module dask.array), 219
unpublish_dataset() (distributed.Client method), 782
unravel_index() (in module dask.array), 221
unzip() (dask.bag.Bag method), 387
update() (in module dask.config), 894
upload_file() (distributed.Client method), 782

V
value_counts() (dask.dataframe.Series method), 675
values (dask.dataframe.DataFrame attribute), 568
values (dask.dataframe.Series attribute), 677
values() (dask.highlevelgraph.HighLevelGraph method), 842
var() (dask.array.Array method), 341
var() (dask.bag.Bag method), 388
var() (dask.dataframe.DataFrame method), 568

W
wait() (in module distributed), 762
wald() (in module dask.array.random), 304
weibull() (in module dask.array.random), 304
where() (dask.dataframe.DataFrame method), 569
where() (dask.dataframe.Series method), 678
who_has() (distributed.Client method), 782
write_scheduler_file() (distributed.Client method), 783

Z
zeros() (in module dask.array), 226
zeros_like() (in module dask.array), 227
zip() (in module dask.bag), 392
zipf() (in module dask.array.random), 305