Zarr-Python

Release 2.17.1

Zarr Developers

Mar 06, 2024

CONTENTS

CHAPTER

ONE

GETTING STARTED

Zarr is a format for the storage of chunked, compressed, N-dimensional arrays inspired by HDF5, h5py and bcolz.

The project is fiscally sponsored by NumFOCUS, a US 501(c)(3) public charity, and development is supported by the MRC Centre for Genomics and Global Health and the Chan Zuckerberg Initiative.

These documents describe the Zarr Python implementation. More information about the Zarr format can be found on the main website.

1.1 Highlights

- Create N-dimensional arrays with any NumPy dtype.
- Chunk arrays along any dimension.
- Compress and/or filter chunks using any NumCodecs codec.
- Store arrays in memory, on disk, inside a Zip file, on S3, ...
- Read an array concurrently from multiple threads or processes.
- Write to an array concurrently from multiple threads or processes.
- Organize arrays into hierarchies via groups.

1.2 Contributing

Feedback and bug reports are very welcome, please get in touch via the GitHub issue tracker. See *Contributing to Zarr* for further information about contributing to Zarr.

1.3 Projects using Zarr

If you are using Zarr, we would love to hear about it.

1.3.1 Installation

Zarr depends on NumPy. It is generally best to install NumPy first using whatever method is most appropriate for your operating system and Python distribution. Other dependencies should be installed automatically if using one of the installation methods below.

Install Zarr from PyPI:

```
$ pip install zarr
```

Alternatively, install Zarr via conda:

```
$ conda install -c conda-forge zarr
```

To install the latest development version of Zarr, you can use pip with the latest GitHub main:

```
$ pip install git+https://github.com/zarr-developers/zarr-python.git
```

To work with Zarr source code in development, install from GitHub:

```
$ git clone --recursive https://github.com/zarr-developers/zarr-python.git
$ cd zarr-python
$ python -m pip install -e .
```

To verify that Zarr has been fully installed, run the test suite:

```
$ pip install pytest
$ python -m pytest -v --pyargs zarr
```

CHAPTER

TWO

TUTORIAL

Zarr provides classes and functions for working with N-dimensional arrays that behave like NumPy arrays but whose data is divided into chunks and each chunk is compressed. If you are already familiar with HDF5 then Zarr arrays provide similar functionality, but with some additional flexibility.

2.1 Creating an array

Zarr has several functions for creating arrays. For example:

```
>>> import zarr
>>> z = zarr.zeros((10000, 10000), chunks=(1000, 1000), dtype='i4')
>>> z
<zarr.core.Array (10000, 10000) int32>
```

The code above creates a 2-dimensional array of 32-bit integers with 10000 rows and 10000 columns, divided into chunks where each chunk has 1000 rows and 1000 columns (and so there will be 100 chunks in total).

For a complete list of array creation routines see the *zarr.creation* module documentation.

2.2 Reading and writing data

Zarr arrays support a similar interface to NumPy arrays for reading and writing data. For example, the entire array can be filled with a scalar value:

```
>>> z[:] = 42
```

Regions of the array can also be written to, e.g.:

```
>>> import numpy as np
>>> z[0, :] = np.arange(10000)
>>> z[:, 0] = np.arange(10000)
```

The contents of the array can be retrieved by slicing, which will load the requested region into memory as a NumPy array, e.g.:

```
>>> z[0, 0]
0
>>> z[-1, -1]
42
```

(continues on next page)

(continued from previous page)

```
>>> z[0, :]
array([   0,    1,    2, ..., 9997, 9998, 9999], dtype=int32)
>>> z[:, 0]
array([   0,    1,    2, ..., 9997, 9998, 9999], dtype=int32)
>>> z[:]
array([[   0,    1,    2, ..., 9997, 9998, 9999],
       [   1,   42,   42, ...,   42,   42,   42],
       [   2,   42,   42, ...,   42,   42,   42],
       ...,
       [9997,   42,   42, ...,   42,   42,   42],
       [9998,   42,   42, ...,   42,   42,   42],
       [9999,   42,   42, ...,   42,   42,   42]], dtype=int32)
```

2.3 Persistent arrays

In the examples above, compressed data for each chunk of the array was stored in main memory. Zarr arrays can also be stored on a file system, enabling persistence of data between sessions. For example:

```
>>> z1 = zarr.open('data/example.zarr', mode='w', shape=(10000, 10000),
...                chunks=(1000, 1000), dtype='i4')
```

The array above will store its configuration metadata and all compressed chunk data in a directory called 'data/example.zarr' relative to the current working directory. The *zarr.convenience.open()* function provides a convenient way to create a new persistent array or continue working with an existing array. Note that although the function is called "open", there is no need to close an array: data are automatically flushed to disk, and files are automatically closed whenever an array is modified.

Persistent arrays support the same interface for reading and writing data, e.g.:

```
>>> z1[:] = 42
>>> z1[0, :] = np.arange(10000)
>>> z1[:, 0] = np.arange(10000)
```

Check that the data have been written and can be read again:

```
>>> z2 = zarr.open('data/example.zarr', mode='r')
>>> np.all(z1[:] == z2[:])
True
```

If you are just looking for a fast and convenient way to save NumPy arrays to disk then load back into memory later, the functions *zarr.convenience.save()* and *zarr.convenience.load()* may be useful. E.g.:

```
>>> a = np.arange(10)
>>> zarr.save('data/example.zarr', a)
>>> zarr.load('data/example.zarr')
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Please note that there are a number of other options for persistent array storage, see the section on *Storage alternatives* below.

2.4 Resizing and appending

A Zarr array can be resized, which means that any of its dimensions can be increased or decreased in length. For example:

```
>>> z = zarr.zeros(shape=(10000, 10000), chunks=(1000, 1000))
>>> z[:] = 42
>>> z.resize(20000, 10000)
>>> z.shape
(20000, 10000)
```

Note that when an array is resized, the underlying data are not rearranged in any way. If one or more dimensions are shrunk, any chunks falling outside the new array shape will be deleted from the underlying store.

For convenience, Zarr arrays also provide an `append()` method, which can be used to append data to any axis. E.g.:

```
>>> a = np.arange(10000000, dtype='i4').reshape(10000, 1000)
>>> z = zarr.array(a, chunks=(1000, 100))
>>> z.shape
(10000, 1000)
>>> z.append(a)
(20000, 1000)
>>> z.append(np.vstack([a, a]), axis=1)
(20000, 2000)
>>> z.shape
(20000, 2000)
```

2.5 Compressors

A number of different compressors can be used with Zarr. A separate package called NumCodecs is available which provides a common interface to various compressor libraries including Blosc, Zstandard, LZ4, Zlib, BZ2 and LZMA. Different compressors can be provided via the `compressor` keyword argument accepted by all array creation functions. For example:

```
>>> from numcodecs import Blosc
>>> compressor = Blosc(cname='zstd', clevel=3, shuffle=Blosc.BITSHUFFLE)
>>> data = np.arange(100000000, dtype='i4').reshape(10000, 10000)
>>> z = zarr.array(data, chunks=(1000, 1000), compressor=compressor)
>>> z.compressor
Blosc(cname='zstd', clevel=3, shuffle=BITSHUFFLE, blocksize=0)
```

This array above will use Blosc as the primary compressor, using the Zstandard algorithm (compression level 3) internally within Blosc, and with the bit-shuffle filter applied.

When using a compressor, it can be useful to get some diagnostics on the compression ratio. Zarr arrays provide a `info` property which can be used to print some diagnostics, e.g.:

```
>>> z.info
Type               : zarr.core.Array
Data type          : int32
Shape              : (10000, 10000)
Chunk shape        : (1000, 1000)
```

(continues on next page)

(continued from previous page)

```
Order               : C
Read-only           : False
Compressor          : Blosc(cname='zstd', clevel=3, shuffle=BITSHUFFLE,
                    : blocksize=0)
Store type          : zarr.storage.KVStore
No. bytes           : 400000000 (381.5M)
No. bytes stored    : 3379344 (3.2M)
Storage ratio       : 118.4
Chunks initialized : 100/100
```

If you don't specify a compressor, by default Zarr uses the Blosc compressor. Blosc is generally very fast and can be configured in a variety of ways to improve the compression ratio for different types of data. Blosc is in fact a "meta-compressor", which means that it can use a number of different compression algorithms internally to compress the data. Blosc also provides highly optimized implementations of byte- and bit-shuffle filters, which can improve compression ratios for some data. A list of the internal compression libraries available within Blosc can be obtained via:

```
>>> from numcodecs import blosc
>>> blosc.list_compressors()
['blosclz', 'lz4', 'lz4hc', 'snappy', 'zlib', 'zstd']
```

In addition to Blosc, other compression libraries can also be used. For example, here is an array using Zstandard compression, level 1:

```
>>> from numcodecs import Zstd
>>> z = zarr.array(np.arange(100000000, dtype='i4').reshape(10000, 10000),
...                chunks=(1000, 1000), compressor=Zstd(level=1))
>>> z.compressor
Zstd(level=1)
```

Here is an example using LZMA with a custom filter pipeline including LZMA's built-in delta filter:

```
>>> import lzma
>>> lzma_filters = [dict(id=lzma.FILTER_DELTA, dist=4),
...                 dict(id=lzma.FILTER_LZMA2, preset=1)]
>>> from numcodecs import LZMA
>>> compressor = LZMA(filters=lzma_filters)
>>> z = zarr.array(np.arange(100000000, dtype='i4').reshape(10000, 10000),
...                chunks=(1000, 1000), compressor=compressor)
>>> z.compressor
LZMA(format=1, check=-1, preset=None, filters=[{'dist': 4, 'id': 3}, {'id': 33, 'preset
↪': 1}])
```

The default compressor can be changed by setting the value of the `zarr.storage.default_compressor` variable, e.g.:

```
>>> import zarr.storage
>>> from numcodecs import Zstd, Blosc
>>> # switch to using Zstandard
... zarr.storage.default_compressor = Zstd(level=1)
>>> z = zarr.zeros(100000000, chunks=1000000)
>>> z.compressor
Zstd(level=1)
```

(continues on next page)

(continued from previous page)

```
>>> # switch back to Blosc defaults
... zarr.storage.default_compressor = Blosc()
```

To disable compression, set `compressor=None` when creating an array, e.g.:

```
>>> z = zarr.zeros(100000000, chunks=1000000, compressor=None)
>>> z.compressor is None
True
```

2.6 Filters

In some cases, compression can be improved by transforming the data in some way. For example, if nearby values tend to be correlated, then shuffling the bytes within each numerical value or storing the difference between adjacent values may increase compression ratio. Some compressors provide built-in filters that apply transformations to the data prior to compression. For example, the Blosc compressor has built-in implementations of byte- and bit-shuffle filters, and the LZMA compressor has a built-in implementation of a delta filter. However, to provide additional flexibility for implementing and using filters in combination with different compressors, Zarr also provides a mechanism for configuring filters outside of the primary compressor.

Here is an example using a delta filter with the Blosc compressor:

```
>>> from numcodecs import Blosc, Delta
>>> filters = [Delta(dtype='i4')]
>>> compressor = Blosc(cname='zstd', clevel=1, shuffle=Blosc.SHUFFLE)
>>> data = np.arange(100000000, dtype='i4').reshape(10000, 10000)
>>> z = zarr.array(data, chunks=(1000, 1000), filters=filters, compressor=compressor)
>>> z.info
Type               : zarr.core.Array
Data type          : int32
Shape              : (10000, 10000)
Chunk shape        : (1000, 1000)
Order              : C
Read-only          : False
Filter [0]         : Delta(dtype='<i4')
Compressor         : Blosc(cname='zstd', clevel=1, shuffle=SHUFFLE, blocksize=0)
Store type         : zarr.storage.KVStore
No. bytes          : 400000000 (381.5M)
No. bytes stored   : 1290562 (1.2M)
Storage ratio      : 309.9
Chunks initialized : 100/100
```

For more information about available filter codecs, see the Numcodecs documentation.

2.7 Groups

Zarr supports hierarchical organization of arrays via groups. As with arrays, groups can be stored in memory, on disk, or via other storage systems that support a similar interface.

To create a group, use the `zarr.group()` function:

```
>>> root = zarr.group()
>>> root
<zarr.hierarchy.Group '/'>
```

Groups have a similar API to the Group class from h5py. For example, groups can contain other groups:

```
>>> foo = root.create_group('foo')
>>> bar = foo.create_group('bar')
```

Groups can also contain arrays, e.g.:

```
>>> z1 = bar.zeros('baz', shape=(10000, 10000), chunks=(1000, 1000), dtype='i4')
>>> z1
<zarr.core.Array '/foo/bar/baz' (10000, 10000) int32>
```

Arrays are known as "datasets" in HDF5 terminology. For compatibility with h5py, Zarr groups also implement the `create_dataset()` and `require_dataset()` methods, e.g.:

```
>>> z = bar.create_dataset('quux', shape=(10000, 10000), chunks=(1000, 1000), dtype='i4')
>>> z
<zarr.core.Array '/foo/bar/quux' (10000, 10000) int32>
```

Members of a group can be accessed via the suffix notation, e.g.:

```
>>> root['foo']
<zarr.hierarchy.Group '/foo'>
```

The '/' character can be used to access multiple levels of the hierarchy in one call, e.g.:

```
>>> root['foo/bar']
<zarr.hierarchy.Group '/foo/bar'>
>>> root['foo/bar/baz']
<zarr.core.Array '/foo/bar/baz' (10000, 10000) int32>
```

The *zarr.hierarchy.Group.tree()* method can be used to print a tree representation of the hierarchy, e.g.:

```
>>> root.tree()
/
 └── foo
     └── bar
         ├── baz (10000, 10000) int32
         └── quux (10000, 10000) int32
```

The *zarr.convenience.open()* function provides a convenient way to create or re-open a group stored in a directory on the file-system, with sub-groups stored in sub-directories, e.g.:

```
>>> root = zarr.open('data/group.zarr', mode='w')
>>> root
```

(continues on next page)

(continued from previous page)

```
<zarr.hierarchy.Group '/'>
>>> z = root.zeros('foo/bar/baz', shape=(10000, 10000), chunks=(1000, 1000), dtype='i4')
>>> z
<zarr.core.Array '/foo/bar/baz' (10000, 10000) int32>
```

Groups can be used as context managers (in a `with` statement). If the underlying store has a `close` method, it will be called on exit.

For more information on groups see the *zarr.hierarchy* and *zarr.convenience* API docs.

2.8 Array and group diagnostics

Diagnostic information about arrays and groups is available via the `info` property. E.g.:

```
>>> root = zarr.group()
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=1000000, chunks=100000, dtype='i8')
>>> bar[:] = 42
>>> baz = foo.zeros('baz', shape=(1000, 1000), chunks=(100, 100), dtype='f4')
>>> baz[:] = 4.2
>>> root.info
Name        : /
Type        : zarr.hierarchy.Group
Read-only   : False
Store type  : zarr.storage.MemoryStore
No. members : 1
No. arrays  : 0
No. groups  : 1
Groups      : foo

>>> foo.info
Name        : /foo
Type        : zarr.hierarchy.Group
Read-only   : False
Store type  : zarr.storage.MemoryStore
No. members : 2
No. arrays  : 2
No. groups  : 0
Arrays      : bar, baz

>>> bar.info
Name               : /foo/bar
Type               : zarr.core.Array
Data type          : int64
Shape              : (1000000,)
Chunk shape        : (100000,)
Order              : C
Read-only          : False
Compressor         : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type         : zarr.storage.MemoryStore
No. bytes          : 8000000 (7.6M)
```

(continues on next page)

(continued from previous page)

```
No. bytes stored   : 33240 (32.5K)
Storage ratio      : 240.7
Chunks initialized : 10/10

>>> baz.info
Name               : /foo/baz
Type               : zarr.core.Array
Data type          : float32
Shape              : (1000, 1000)
Chunk shape        : (100, 100)
Order              : C
Read-only          : False
Compressor         : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type         : zarr.storage.MemoryStore
No. bytes          : 4000000 (3.8M)
No. bytes stored   : 23943 (23.4K)
Storage ratio      : 167.1
Chunks initialized : 100/100
```

Groups also have the *zarr.hierarchy.Group.tree()* method, e.g.:

```
>>> root.tree()
/
 └── foo
     ├── bar (1000000,) int64
     └── baz (1000, 1000) float32
```

If you're using Zarr within a Jupyter notebook (requires ipytree), calling `tree()` will generate an interactive tree representation, see the repr_tree.ipynb notebook for more examples.

2.9 User attributes

Zarr arrays and groups support custom key/value attributes, which can be useful for storing application-specific metadata. For example:

```
>>> root = zarr.group()
>>> root.attrs['foo'] = 'bar'
>>> z = root.zeros('zzz', shape=(10000, 10000))
>>> z.attrs['baz'] = 42
>>> z.attrs['qux'] = [1, 4, 7, 12]
>>> sorted(root.attrs)
['foo']
>>> 'foo' in root.attrs
True
>>> root.attrs['foo']
'bar'
>>> sorted(z.attrs)
['baz', 'qux']
>>> z.attrs['baz']
42
```

(continues on next page)

(continued from previous page)

```
>>> z.attrs['qux']
[1, 4, 7, 12]
```

Internally Zarr uses JSON to store array attributes, so attribute values must be JSON serializable.

2.10 Advanced indexing

As of version 2.2, Zarr arrays support several methods for advanced or "fancy" indexing, which enable a subset of data items to be extracted or updated in an array without loading the entire array into memory.

Note that although this functionality is similar to some of the advanced indexing capabilities available on NumPy arrays and on h5py datasets, **the Zarr API for advanced indexing is different from both NumPy and h5py**, so please read this section carefully. For a complete description of the indexing API, see the documentation for the *zarr.core.Array* class.

2.10.1 Indexing with coordinate arrays

Items from a Zarr array can be extracted by providing an integer array of coordinates. E.g.:

```
>>> z = zarr.array(np.arange(10) ** 2)
>>> z[:]
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
>>> z.get_coordinate_selection([2, 5])
array([ 4, 25])
```

Coordinate arrays can also be used to update data, e.g.:

```
>>> z.set_coordinate_selection([2, 5], [-1, -2])
>>> z[:]
array([ 0,  1, -1,  9, 16, -2, 36, 49, 64, 81])
```

For multidimensional arrays, coordinates must be provided for each dimension, e.g.:

```
>>> z = zarr.array(np.arange(15).reshape(3, 5))
>>> z[:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> z.get_coordinate_selection(([0, 2], [1, 3]))
array([ 1, 13])
>>> z.set_coordinate_selection(([0, 2], [1, 3]), [-1, -2])
>>> z[:]
array([[ 0, -1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, -2, 14]])
```

For convenience, coordinate indexing is also available via the `vindex` property, as well as the square bracket operator, e.g.:

```
>>> z.vindex[[0, 2], [1, 3]]
array([-1, -2])
>>> z.vindex[[0, 2], [1, 3]] = [-3, -4]
>>> z[:]
array([[ 0, -3,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, -4, 14]])
>>> z[[0, 2], [1, 3]]
array([-3, -4])
```

When the indexing arrays have different shapes, they are broadcast together. That is, the following two calls are equivalent:

```
>>> z[1, [1, 3]]
array([6, 8])
>>> z[[1, 1], [1, 3]]
array([6, 8])
```

2.10.2 Indexing with a mask array

Items can also be extracted by providing a Boolean mask. E.g.:

```
>>> z = zarr.array(np.arange(10) ** 2)
>>> z[:]
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
>>> sel = np.zeros_like(z, dtype=bool)
>>> sel[2] = True
>>> sel[5] = True
>>> z.get_mask_selection(sel)
array([ 4, 25])
>>> z.set_mask_selection(sel, [-1, -2])
>>> z[:]
array([ 0,  1, -1,  9, 16, -2, 36, 49, 64, 81])
```

Here's a multidimensional example:

```
>>> z = zarr.array(np.arange(15).reshape(3, 5))
>>> z[:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> sel = np.zeros_like(z, dtype=bool)
>>> sel[0, 1] = True
>>> sel[2, 3] = True
>>> z.get_mask_selection(sel)
array([ 1, 13])
>>> z.set_mask_selection(sel, [-1, -2])
>>> z[:]
array([[ 0, -1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, -2, 14]])
```

For convenience, mask indexing is also available via the `vindex` property, e.g.:

```
>>> z.vindex[sel]
array([-1, -2])
>>> z.vindex[sel] = [-3, -4]
>>> z[:]
array([[ 0, -3,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, -4, 14]])
```

Mask indexing is conceptually the same as coordinate indexing, and is implemented internally via the same machinery. Both styles of indexing allow selecting arbitrary items from an array, also known as point selection.

2.10.3 Orthogonal indexing

Zarr arrays also support methods for orthogonal indexing, which allows selections to be made along each dimension of an array independently. For example, this allows selecting a subset of rows and/or columns from a 2-dimensional array. E.g.:

```
>>> z = zarr.array(np.arange(15).reshape(3, 5))
>>> z[:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> z.get_orthogonal_selection(([0, 2], slice(None)))  # select first and third rows
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])
>>> z.get_orthogonal_selection((slice(None), [1, 3]))  # select second and fourth columns
array([[ 1,  3],
       [ 6,  8],
       [11, 13]])
>>> z.get_orthogonal_selection(([0, 2], [1, 3]))       # select rows [0, 2] and columns␣
↪[1, 4]
array([[ 1,  3],
       [11, 13]])
```

Data can also be modified, e.g.:

```
>>> z.set_orthogonal_selection(([0, 2], [1, 3]), [[-1, -2], [-3, -4]])
>>> z[:]
array([[ 0, -1,  2, -2,  4],
       [ 5,  6,  7,  8,  9],
       [10, -3, 12, -4, 14]])
```

For convenience, the orthogonal indexing functionality is also available via the `oindex` property, e.g.:

```
>>> z = zarr.array(np.arange(15).reshape(3, 5))
>>> z.oindex[[0, 2], :]  # select first and third rows
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])
>>> z.oindex[:, [1, 3]]  # select second and fourth columns
array([[ 1,  3],
       [ 6,  8],
       [11, 13]])
```

(continues on next page)

(continued from previous page)

```
>>> z.oindex[[0, 2], [1, 3]]  # select rows [0, 2] and columns [1, 4]
array([[ 1,  3],
       [11, 13]])
>>> z.oindex[[0, 2], [1, 3]] = [[-1, -2], [-3, -4]]
>>> z[:]
array([[ 0, -1,  2, -2,  4],
       [ 5,  6,  7,  8,  9],
       [10, -3, 12, -4, 14]])
```

Any combination of integer, slice, 1D integer array and/or 1D Boolean array can be used for orthogonal indexing.

If the index contains at most one iterable, and otherwise contains only slices and integers, orthogonal indexing is also available directly on the array:

```
>>> z = zarr.array(np.arange(15).reshape(3, 5))
>>> all(z.oindex[[0, 2], :] == z[[0, 2], :])
True
```

2.10.4 Block Indexing

As of version 2.16.0, Zarr also support block indexing, which allows selections of whole chunks based on their logical indices along each dimension of an array. For example, this allows selecting a subset of chunk aligned rows and/or columns from a 2-dimensional array. E.g.:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.array(np.arange(100).reshape(10, 10), chunks=(3, 3))
```

Retrieve items by specifying their block coordinates:

```
>>> z.get_block_selection(1)
array([[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

Equivalent slicing:

```
>>> z[3:6]
array([[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

For convenience, the block selection functionality is also available via the *blocks* property, e.g.:

```
>>> z.blocks[1]
array([[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

Block index arrays may be multidimensional to index multidimensional arrays. For example:

```
>>> z.blocks[0, 1:3]
array([[ 3,  4,  5,  6,  7,  8],
       [13, 14, 15, 16, 17, 18],
       [23, 24, 25, 26, 27, 28]])
```

Data can also be modified. Let's start by a simple 2D array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.zeros((6, 6), dtype=int, chunks=2)
```

Set data for a selection of items:

```
>>> z.set_block_selection((1, 0), 1)
>>> z[...]
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
```

For convenience, this functionality is also available via the `blocks` property. E.g.:

```
>>> z.blocks[:, 2] = 7
>>> z[...]
array([[0, 0, 0, 0, 7, 7],
       [0, 0, 0, 0, 7, 7],
       [1, 1, 0, 0, 7, 7],
       [1, 1, 0, 0, 7, 7],
       [0, 0, 0, 0, 7, 7],
       [0, 0, 0, 0, 7, 7]])
```

Any combination of integer and slice can be used for block indexing:

```
>>> z.blocks[2, 1:3]
array([[0, 0, 7, 7],
       [0, 0, 7, 7]])
```

2.10.5 Indexing fields in structured arrays

All selection methods support a `fields` parameter which allows retrieving or replacing data for a specific field in an array with a structured dtype. E.g.:

```
>>> a = np.array([(b'aaa', 1, 4.2),
...               (b'bbb', 2, 8.4),
...               (b'ccc', 3, 12.6)],
...              dtype=[('foo', 'S3'), ('bar', 'i4'), ('baz', 'f8')])
>>> z = zarr.array(a)
>>> z['foo']
array([b'aaa', b'bbb', b'ccc'],
      dtype='|S3')
```

(continues on next page)

(continued from previous page)

```
>>> z['baz']
array([  4.2,   8.4,  12.6])
>>> z.get_basic_selection(slice(0, 2), fields='bar')
array([1, 2], dtype=int32)
>>> z.get_coordinate_selection([0, 2], fields=['foo', 'baz'])
array([(b'aaa',   4.2), (b'ccc',  12.6)],
      dtype=[('foo', 'S3'), ('baz', '<f8')])
```

2.11 Storage alternatives

Zarr can use any object that implements the `MutableMapping` interface from the `collections` module in the Python standard library as the store for a group or an array.

Some pre-defined storage classes are provided in the *zarr.storage* module. For example, the *zarr.storage.DirectoryStore* class provides a `MutableMapping` interface to a directory on the local file system. This is used under the hood by the *zarr.convenience.open()* function. In other words, the following code:

```
>>> z = zarr.open('data/example.zarr', mode='w', shape=1000000, dtype='i4')
```

...is short-hand for:

```
>>> store = zarr.DirectoryStore('data/example.zarr')
>>> z = zarr.create(store=store, overwrite=True, shape=1000000, dtype='i4')
```

...and the following code:

```
>>> root = zarr.open('data/example.zarr', mode='w')
```

...is short-hand for:

```
>>> store = zarr.DirectoryStore('data/example.zarr')
>>> root = zarr.group(store=store, overwrite=True)
```

Any other compatible storage class could be used in place of *zarr.storage.DirectoryStore* in the code examples above. For example, here is an array stored directly into a ZIP archive, via the *zarr.storage.ZipStore* class:

```
>>> store = zarr.ZipStore('data/example.zip', mode='w')
>>> root = zarr.group(store=store)
>>> z = root.zeros('foo/bar', shape=(1000, 1000), chunks=(100, 100), dtype='i4')
>>> z[:] = 42
>>> store.close()
```

Re-open and check that data have been written:

```
>>> store = zarr.ZipStore('data/example.zip', mode='r')
>>> root = zarr.group(store=store)
>>> z = root['foo/bar']
>>> z[:]
array([[42, 42, 42, ..., 42, 42, 42],
       [42, 42, 42, ..., 42, 42, 42],
       [42, 42, 42, ..., 42, 42, 42],
```

(continues on next page)

(continued from previous page)

```
       ...,
       [42, 42, 42, ..., 42, 42, 42],
       [42, 42, 42, ..., 42, 42, 42],
       [42, 42, 42, ..., 42, 42, 42]], dtype=int32)
>>> store.close()
```

Note that there are some limitations on how ZIP archives can be used, because items within a ZIP archive cannot be updated in place. This means that data in the array should only be written once and write operations should be aligned with chunk boundaries. Note also that the `close()` method must be called after writing any data to the store, otherwise essential records will not be written to the underlying ZIP archive.

Another storage alternative is the *zarr.storage.DBMStore* class, added in Zarr version 2.2. This class allows any DBM-style database to be used for storing an array or group. Here is an example using a Berkeley DB B-tree database for storage (requires bsddb3 to be installed):

```
>>> import bsddb3
>>> store = zarr.DBMStore('data/example.bdb', open=bsddb3.btopen)
>>> root = zarr.group(store=store, overwrite=True)
>>> z = root.zeros('foo/bar', shape=(1000, 1000), chunks=(100, 100), dtype='i4')
>>> z[:] = 42
>>> store.close()
```

Also added in Zarr version 2.2 is the *zarr.storage.LMDBStore* class which enables the lightning memory-mapped database (LMDB) to be used for storing an array or group (requires lmdb to be installed):

```
>>> store = zarr.LMDBStore('data/example.lmdb')
>>> root = zarr.group(store=store, overwrite=True)
>>> z = root.zeros('foo/bar', shape=(1000, 1000), chunks=(100, 100), dtype='i4')
>>> z[:] = 42
>>> store.close()
```

In Zarr version 2.3 is the *zarr.storage.SQLiteStore* class which enables the SQLite database to be used for storing an array or group (requires Python is built with SQLite support):

```
>>> store = zarr.SQLiteStore('data/example.sqldb')
>>> root = zarr.group(store=store, overwrite=True)
>>> z = root.zeros('foo/bar', shape=(1000, 1000), chunks=(100, 100), dtype='i4')
>>> z[:] = 42
>>> store.close()
```

Also added in Zarr version 2.3 are two storage classes for interfacing with server-client databases. The *zarr.storage.RedisStore* class interfaces Redis (an in memory data structure store), and the `zarr.storage.MongoDB` class interfaces with MongoDB (an object oriented NoSQL database). These stores respectively require the redis-py and pymongo packages to be installed.

For compatibility with the N5 data format, Zarr also provides an N5 backend (this is currently an experimental feature). Similar to the ZIP storage class, an *zarr.n5.N5Store* can be instantiated directly:

```
>>> store = zarr.N5Store('data/example.n5')
>>> root = zarr.group(store=store)
>>> z = root.zeros('foo/bar', shape=(1000, 1000), chunks=(100, 100), dtype='i4')
>>> z[:] = 42
```

For convenience, the N5 backend will automatically be chosen when the filename ends with *.n5*:

```
>>> root = zarr.open('data/example.n5', mode='w')
```

2.11.1 Distributed/cloud storage

It is also possible to use distributed storage systems. The Dask project has implementations of the `MutableMapping` interface for Amazon S3 (S3Map), Hadoop Distributed File System (HDFSMap) and Google Cloud Storage (GCSMap), which can be used with Zarr.

Here is an example using S3Map to read an array created previously:

```
>>> import s3fs
>>> import zarr
>>> s3 = s3fs.S3FileSystem(anon=True, client_kwargs=dict(region_name='eu-west-2'))
>>> store = s3fs.S3Map(root='zarr-demo/store', s3=s3, check=False)
>>> root = zarr.group(store=store)
>>> z = root['foo/bar/baz']
>>> z
<zarr.core.Array '/foo/bar/baz' (21,) |S1>
>>> z.info
Name               : /foo/bar/baz
Type               : zarr.core.Array
Data type          : |S1
Shape              : (21,)
Chunk shape        : (7,)
Order              : C
Read-only          : False
Compressor         : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type         : zarr.storage.KVStore
No. bytes          : 21
No. bytes stored   : 382
Storage ratio      : 0.1
Chunks initialized : 3/3
>>> z[:]
array([b'H', b'e', b'l', b'l', b'o', b' ', b'f', b'r', b'o', b'm', b' ',
       b't', b'h', b'e', b' ', b'c', b'l', b'o', b'u', b'd', b'!'],
      dtype='|S1')
>>> z[:].tobytes()
b'Hello from the cloud!'
```

Zarr now also has a builtin storage backend for Azure Blob Storage. The class is *zarr.storage.ABSStore* (requires azure-storage-blob to be installed):

```
>>> import azure.storage.blob
>>> container_client = azure.storage.blob.ContainerClient(...)
>>> store = zarr.ABSStore(client=container_client, prefix='zarr-testing')
>>> root = zarr.group(store=store, overwrite=True)
>>> z = root.zeros('foo/bar', shape=(1000, 1000), chunks=(100, 100), dtype='i4')
>>> z[:] = 42
```

When using an actual storage account, provide `account_name` and `account_key` arguments to *zarr.storage.ABSStore*, the above client is just testing against the emulator. Please also note that this is an experimental feature.

Note that retrieving data from a remote service via the network can be significantly slower than retrieving data from a local file system, and will depend on network latency and bandwidth between the client and server systems. If you

are experiencing poor performance, there are several things you can try. One option is to increase the array chunk size, which will reduce the number of chunks and thus reduce the number of network round-trips required to retrieve data for an array (and thus reduce the impact of network latency). Another option is to try to increase the compression ratio by changing compression options or trying a different compressor (which will reduce the impact of limited network bandwidth).

As of version 2.2, Zarr also provides the *zarr.storage.LRUStoreCache* which can be used to implement a local in-memory cache layer over a remote store. E.g.:

```
>>> s3 = s3fs.S3FileSystem(anon=True, client_kwargs=dict(region_name='eu-west-2'))
>>> store = s3fs.S3Map(root='zarr-demo/store', s3=s3, check=False)
>>> cache = zarr.LRUStoreCache(store, max_size=2**28)
>>> root = zarr.group(store=cache)
>>> z = root['foo/bar/baz']
>>> from timeit import timeit
>>> # first data access is relatively slow, retrieved from store
... timeit('print(z[:].tobytes())', number=1, globals=globals())
b'Hello from the cloud!'
0.1081731989979744
>>> # second data access is faster, uses cache
... timeit('print(z[:].tobytes())', number=1, globals=globals())
b'Hello from the cloud!'
0.0009490990014455747
```

If you are still experiencing poor performance with distributed/cloud storage, please raise an issue on the GitHub issue tracker with any profiling data you can provide, as there may be opportunities to optimise further either within Zarr or within the mapping interface to the storage.

2.11.2 IO with fsspec

As of version 2.5, zarr supports passing URLs directly to fsspec, and having it create the "mapping" instance automatically. This means, that for all of the backend storage implementations supported by fsspec, you can skip importing and configuring the storage explicitly. For example:

```
>>> g = zarr.open_group("s3://zarr-demo/store", storage_options={'anon': True})
>>> g['foo/bar/baz'][:].tobytes()
b'Hello from the cloud!'
```

The provision of the protocol specifier "s3://" will select the correct backend. Notice the kwargs `storage_options`, used to pass parameters to that backend.

As of version 2.6, write mode and complex URLs are also supported, such as:

```
>>> g = zarr.open_group("simplecache::s3://zarr-demo/store",
...                     storage_options={"s3": {'anon': True}})
>>> g['foo/bar/baz'][:].tobytes()  # downloads target file
b'Hello from the cloud!'
>>> g['foo/bar/baz'][:].tobytes()  # uses cached file
b'Hello from the cloud!'
```

The second invocation here will be much faster. Note that the `storage_options` have become more complex here, to account for the two parts of the supplied URL.

It is also possible to initialize the filesystem outside of Zarr and then pass it through. This requires creating an *zarr.storage.FSStore* object explicitly. For example:

```
>>> import s3fs
>>> fs = s3fs.S3FileSystem(anon=True)
>>> store = zarr.storage.FSStore('/zarr-demo/store', fs=fs)
>>> g = zarr.open_group(store)
```

This is useful in cases where you want to also use the same fsspec filesystem object separately from Zarr.

2.11.3 Accessing ZIP archives on S3

The built-in *zarr.storage.ZipStore* will only work with paths on the local file-system; however it is possible to access ZIP-archived Zarr data on the cloud via the ZipFileSystem class from `fsspec`. The following example demonstrates how to access a ZIP-archived Zarr group on s3 using s3fs and `ZipFileSystem`:

```
>>> s3_path = "s3://path/to/my.zarr.zip"
>>>
>>> s3 = s3fs.S3FileSystem()
>>> f = s3.open(s3_path)
>>> fs = ZipFileSystem(f, mode="r")
>>> store = FSMap("", fs, check=False)
>>>
>>> # caching may improve performance when repeatedly reading the same data
>>> cache = zarr.storage.LRUStoreCache(store, max_size=2**28)
>>> z = zarr.group(store=cache)
```

This store can also be generated with `fsspec`'s handler chaining, like so:

```
>>> store = zarr.storage.FSStore(url=f"zip::{s3_path}",  mode="r")
```

This can be especially useful if you have a very large ZIP-archived Zarr array or group on s3 and only need to access a small portion of it.

2.11.4 Consolidating metadata

Since there is a significant overhead for every connection to a cloud object store such as S3, the pattern described in the previous section may incur significant latency while scanning the metadata of the array hierarchy, even though each individual metadata object is small. For cases such as these, once the data are static and can be regarded as read-only, at least for the metadata/structure of the array hierarchy, the many metadata objects can be consolidated into a single one via *zarr.convenience.consolidate_metadata()*. Doing this can greatly increase the speed of reading the array metadata, e.g.:

```
>>> zarr.consolidate_metadata(store)
```

This creates a special key with a copy of all of the metadata from all of the metadata objects in the store.

Later, to open a Zarr store with consolidated metadata, use *zarr.convenience.open_consolidated()*, e.g.:

```
>>> root = zarr.open_consolidated(store)
```

This uses the special key to read all of the metadata in a single call to the backend storage.

Note that, the hierarchy could still be opened in the normal way and altered, causing the consolidated metadata to become out of sync with the real state of the array hierarchy. In this case, *zarr.convenience.consolidate_metadata()* would need to be called again.

To protect against consolidated metadata accidentally getting out of sync, the root group returned by *zarr.convenience.open_consolidated()* is read-only for the metadata, meaning that no new groups or arrays can be created, and arrays cannot be resized. However, data values with arrays can still be updated.

2.12 Copying/migrating data

If you have some data in an HDF5 file and would like to copy some or all of it into a Zarr group, or vice-versa, the *zarr.convenience.copy()* and *zarr.convenience.copy_all()* functions can be used. Here's an example copying a group named 'foo' from an HDF5 file to a Zarr group:

```
>>> import h5py
>>> import zarr
>>> import numpy as np
>>> source = h5py.File('data/example.h5', mode='w')
>>> foo = source.create_group('foo')
>>> baz = foo.create_dataset('bar/baz', data=np.arange(100), chunks=(50,))
>>> spam = source.create_dataset('spam', data=np.arange(100, 200), chunks=(30,))
>>> zarr.tree(source)
/
 ├── foo
 │   └── bar
 │       └── baz (100,) int64
 └── spam (100,) int64
>>> dest = zarr.open_group('data/example.zarr', mode='w')
>>> from sys import stdout
>>> zarr.copy(source['foo'], dest, log=stdout)
copy /foo
copy /foo/bar
copy /foo/bar/baz (100,) int64
all done: 3 copied, 0 skipped, 800 bytes copied
(3, 0, 800)
>>> dest.tree()  # N.B., no spam
/
 └── foo
     └── bar
         └── baz (100,) int64
>>> source.close()
```

If rather than copying a single group or array you would like to copy all groups and arrays, use *zarr.convenience.copy_all()*, e.g.:

```
>>> source = h5py.File('data/example.h5', mode='r')
>>> dest = zarr.open_group('data/example2.zarr', mode='w')
>>> zarr.copy_all(source, dest, log=stdout)
copy /foo
copy /foo/bar
copy /foo/bar/baz (100,) int64
copy /spam (100,) int64
all done: 4 copied, 0 skipped, 1,600 bytes copied
(4, 0, 1600)
>>> dest.tree()
/
```

(continues on next page)

(continued from previous page)

```
 ├── foo
 │    └── bar
 │        └── baz (100,) int64
 └── spam (100,) int64
```

If you need to copy data between two Zarr groups, the *zarr.convenience.copy()* and *zarr.convenience.copy_all()* functions can be used and provide the most flexibility. However, if you want to copy data in the most efficient way possible, without changing any configuration options, the *zarr.convenience.copy_store()* function can be used. This function copies data directly between the underlying stores, without any decompression or re-compression, and so should be faster. E.g.:

```
>>> import zarr
>>> import numpy as np
>>> store1 = zarr.DirectoryStore('data/example.zarr')
>>> root = zarr.group(store1, overwrite=True)
>>> baz = root.create_dataset('foo/bar/baz', data=np.arange(100), chunks=(50,))
>>> spam = root.create_dataset('spam', data=np.arange(100, 200), chunks=(30,))
>>> root.tree()
/
 ├── foo
 │    └── bar
 │        └── baz (100,) int64
 └── spam (100,) int64
>>> from sys import stdout
>>> store2 = zarr.ZipStore('data/example.zip', mode='w')
>>> zarr.copy_store(store1, store2, log=stdout)
copy .zgroup
copy foo/.zgroup
copy foo/bar/.zgroup
copy foo/bar/baz/.zarray
copy foo/bar/baz/0
copy foo/bar/baz/1
copy spam/.zarray
copy spam/0
copy spam/1
copy spam/2
copy spam/3
all done: 11 copied, 0 skipped, 1,138 bytes copied
(11, 0, 1138)
>>> new_root = zarr.group(store2)
>>> new_root.tree()
/
 ├── foo
 │    └── bar
 │        └── baz (100,) int64
 └── spam (100,) int64
>>> new_root['foo/bar/baz'][:]
array([ 0,  1,  2,  ..., 97, 98, 99])
>>> store2.close()  # ZIP stores need to be closed
```

2.13 String arrays

There are several options for storing arrays of strings.

If your strings are all ASCII strings, and you know the maximum length of the string in your array, then you can use an array with a fixed-length bytes dtype. E.g.:

```
>>> z = zarr.zeros(10, dtype='S6')
>>> z
<zarr.core.Array (10,) |S6>
>>> z[0] = b'Hello'
>>> z[1] = b'world!'
>>> z[:]
array([b'Hello', b'world!', b'', b'', b'', b'', b'', b'', b'', b''],
      dtype='|S6')
```

A fixed-length unicode dtype is also available, e.g.:

```
>>> greetings = ['¡Hola mundo!', 'Hej Världen!', 'Servus Woid!', 'Hei maailma!',
...              'Xin chào th gii', 'Njatjeta Botë!', '  !',
...              '', '', 'Helló, világ!', 'Zdravo svete!',
...              '']
>>> text_data = greetings * 10000
>>> z = zarr.array(text_data, dtype='U20')
>>> z
<zarr.core.Array (120000,) <U20>
>>> z[:]
array(['¡Hola mundo!', 'Hej Världen!', 'Servus Woid!', ...,
       'Helló, világ!', 'Zdravo svete!', ''],
      dtype='<U20')
```

For variable-length strings, the `object` dtype can be used, but a codec must be provided to encode the data (see also *Object arrays* below). At the time of writing there are four codecs available that can encode variable length string objects: `numcodecs.vlen.VLenUTF8`, `numcodecs.json.JSON`, `numcodecs.msgpacks.MsgPack`. and `numcodecs.pickles.Pickle`. E.g. using `VLenUTF8`:

```
>>> import numcodecs
>>> z = zarr.array(text_data, dtype=object, object_codec=numcodecs.VLenUTF8())
>>> z
<zarr.core.Array (120000,) object>
>>> z.filters
[VLenUTF8()]
>>> z[:]
array(['¡Hola mundo!', 'Hej Världen!', 'Servus Woid!', ...,
       'Helló, világ!', 'Zdravo svete!', ''], dtype=object)
```

As a convenience, `dtype=str` (or `dtype=unicode` on Python 2.7) can be used, which is a short-hand for `dtype=object, object_codec=numcodecs.VLenUTF8()`, e.g.:

```
>>> z = zarr.array(text_data, dtype=str)
>>> z
<zarr.core.Array (120000,) object>
>>> z.filters
[VLenUTF8()]
```

(continues on next page)

(continued from previous page)

```
>>> z[:]
array(['¡Hola mundo!', 'Hej Världen!', 'Servus Woid!', ...,
       'Helló, világ!', 'Zdravo svete!', ''], dtype=object)
```

Variable-length byte strings are also supported via `dtype=object`. Again an `object_codec` is required, which can be one of `numcodecs.vlen.VLenBytes` or `numcodecs.pickles.Pickle`. For convenience, `dtype=bytes` (or `dtype=str` on Python 2.7) can be used as a short-hand for `dtype=object, object_codec=numcodecs.VLenBytes()`, e.g.:

```
>>> bytes_data = [g.encode('utf-8') for g in greetings] * 10000
>>> z = zarr.array(bytes_data, dtype=bytes)
>>> z
<zarr.core.Array (120000,) object>
>>> z.filters
[VLenBytes()]
>>> z[:]
array([b'\xc2\xa1Hola mundo!', b'Hej V\xc3\xa4rlden!', b'Servus Woid!',
       ..., b'Hell\xc3\xb3, vil\xc3\xa1g!', b'Zdravo svete!',
       b'\xe0\xb9\x80\xe0\xb8\xae\xe0\xb8\xa5\xe0\xb9\x82\xe0\xb8\xa5\xe0\xb9\x80\xe0\
↪xb8\xa7\xe0\xb8\xb4\xe0\xb8\xa5\xe0\xb8\x94\xe0\xb9\x8c'], dtype=object)
```

If you know ahead of time all the possible string values that can occur, you could also use the `numcodecs.categorize.Categorize` codec to encode each unique string value as an integer. E.g.:

```
>>> categorize = numcodecs.Categorize(greetings, dtype=object)
>>> z = zarr.array(text_data, dtype=object, object_codec=categorize)
>>> z
<zarr.core.Array (120000,) object>
>>> z.filters
[Categorize(dtype='|O', astype='|u1', labels=['¡Hola mundo!', 'Hej Världen!', 'Servus␣
↪Woid!', ...])]
>>> z[:]
array(['¡Hola mundo!', 'Hej Världen!', 'Servus Woid!', ...,
       'Helló, világ!', 'Zdravo svete!', ''], dtype=object)
```

2.14 Object arrays

Zarr supports arrays with an "object" dtype. This allows arrays to contain any type of object, such as variable length unicode strings, or variable length arrays of numbers, or other possibilities. When creating an object array, a codec must be provided via the `object_codec` argument. This codec handles encoding (serialization) of Python objects. The best codec to use will depend on what type of objects are present in the array.

At the time of writing there are three codecs available that can serve as a general purpose object codec and support encoding of a mixture of object types: `numcodecs.json.JSON`, `numcodecs.msgpacks.MsgPack`. and `numcodecs.pickles.Pickle`.

For example, using the JSON codec:

```
>>> z = zarr.empty(5, dtype=object, object_codec=numcodecs.JSON())
>>> z[0] = 42
>>> z[1] = 'foo'
```

(continues on next page)

(continued from previous page)

```
>>> z[2] = ['bar', 'baz', 'qux']
>>> z[3] = {'a': 1, 'b': 2.2}
>>> z[:]
array([42, 'foo', list(['bar', 'baz', 'qux']), {'a': 1, 'b': 2.2}, None], dtype=object)
```

Not all codecs support encoding of all object types. The numcodecs.pickles.Pickle codec is the most flexible, supporting encoding any type of Python object. However, if you are sharing data with anyone other than yourself, then Pickle is not recommended as it is a potential security risk. This is because malicious code can be embedded within pickled data. The JSON and MsgPack codecs do not have any security issues and support encoding of unicode strings, lists and dictionaries. MsgPack is usually faster for both encoding and decoding.

2.14.1 Ragged arrays

If you need to store an array of arrays, where each member array can be of any length and stores the same primitive type (a.k.a. a ragged array), the numcodecs.vlen.VLenArray codec can be used, e.g.:

```
>>> z = zarr.empty(4, dtype=object, object_codec=numcodecs.VLenArray(int))
>>> z
<zarr.core.Array (4,) object>
>>> z.filters
[VLenArray(dtype='<i8')]
>>> z[0] = np.array([1, 3, 5])
>>> z[1] = np.array([4])
>>> z[2] = np.array([7, 9, 14])
>>> z[:]
array([array([1, 3, 5]), array([4]), array([ 7,  9, 14]),
       array([], dtype=int64)], dtype=object)
```

As a convenience, `dtype='array:T'` can be used as a short-hand for `dtype=object, object_codec=numcodecs.VLenArray('T')`, where 'T' can be any NumPy primitive dtype such as 'i4' or 'f8'. E.g.:

```
>>> z = zarr.empty(4, dtype='array:i8')
>>> z
<zarr.core.Array (4,) object>
>>> z.filters
[VLenArray(dtype='<i8')]
>>> z[0] = np.array([1, 3, 5])
>>> z[1] = np.array([4])
>>> z[2] = np.array([7, 9, 14])
>>> z[:]
array([array([1, 3, 5]), array([4]), array([ 7,  9, 14]),
       array([], dtype=int64)], dtype=object)
```

2.15 Chunk optimizations

2.15.1 Chunk size and shape

In general, chunks of at least 1 megabyte (1M) uncompressed size seem to provide better performance, at least when using the Blosc compression library.

The optimal chunk shape will depend on how you want to access the data. E.g., for a 2-dimensional array, if you only ever take slices along the first dimension, then chunk across the second dimension. If you know you want to chunk across an entire dimension you can use `None` or `-1` within the `chunks` argument, e.g.:

```
>>> z1 = zarr.zeros((10000, 10000), chunks=(100, None), dtype='i4')
>>> z1.chunks
(100, 10000)
```

Alternatively, if you only ever take slices along the second dimension, then chunk across the first dimension, e.g.:

```
>>> z2 = zarr.zeros((10000, 10000), chunks=(None, 100), dtype='i4')
>>> z2.chunks
(10000, 100)
```

If you require reasonable performance for both access patterns then you need to find a compromise, e.g.:

```
>>> z3 = zarr.zeros((10000, 10000), chunks=(1000, 1000), dtype='i4')
>>> z3.chunks
(1000, 1000)
```

If you are feeling lazy, you can let Zarr guess a chunk shape for your data by providing `chunks=True`, although please note that the algorithm for guessing a chunk shape is based on simple heuristics and may be far from optimal. E.g.:

```
>>> z4 = zarr.zeros((10000, 10000), chunks=True, dtype='i4')
>>> z4.chunks
(625, 625)
```

If you know you are always going to be loading the entire array into memory, you can turn off chunks by providing `chunks=False`, in which case there will be one single chunk for the array:

```
>>> z5 = zarr.zeros((10000, 10000), chunks=False, dtype='i4')
>>> z5.chunks
(10000, 10000)
```

2.15.2 Chunk memory layout

The order of bytes **within each chunk** of an array can be changed via the `order` keyword argument, to use either C or Fortran layout. For multi-dimensional arrays, these two layouts may provide different compression ratios, depending on the correlation structure within the data. E.g.:

```
>>> a = np.arange(100000000, dtype='i4').reshape(10000, 10000).T
>>> c = zarr.array(a, chunks=(1000, 1000))
>>> c.info
Type               : zarr.core.Array
Data type          : int32
Shape              : (10000, 10000)
```

(continues on next page)

(continued from previous page)

```
Chunk shape       : (1000, 1000)
Order             : C
Read-only         : False
Compressor        : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type        : zarr.storage.KVStore
No. bytes         : 400000000 (381.5M)
No. bytes stored  : 6696010 (6.4M)
Storage ratio     : 59.7
Chunks initialized : 100/100
>>> f = zarr.array(a, chunks=(1000, 1000), order='F')
>>> f.info
Type              : zarr.core.Array
Data type         : int32
Shape             : (10000, 10000)
Chunk shape       : (1000, 1000)
Order             : F
Read-only         : False
Compressor        : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type        : zarr.storage.KVStore
No. bytes         : 400000000 (381.5M)
No. bytes stored  : 4684636 (4.5M)
Storage ratio     : 85.4
Chunks initialized : 100/100
```

In the above example, Fortran order gives a better compression ratio. This is an artificial example but illustrates the general point that changing the order of bytes within chunks of an array may improve the compression ratio, depending on the structure of the data, the compression algorithm used, and which compression filters (e.g., byte-shuffle) have been applied.

2.15.3 Empty chunks

As of version 2.11, it is possible to configure how Zarr handles the storage of chunks that are "empty" (i.e., every element in the chunk is equal to the array's fill value). When creating an array with `write_empty_chunks=False`, Zarr will check whether a chunk is empty before compression and storage. If a chunk is empty, then Zarr does not store it, and instead deletes the chunk from storage if the chunk had been previously stored.

This optimization prevents storing redundant objects and can speed up reads, but the cost is added computation during array writes, since the contents of each chunk must be compared to the fill value, and these advantages are contingent on the content of the array. If you know that your data will form chunks that are almost always non-empty, then there is no advantage to the optimization described above. In this case, creating an array with `write_empty_chunks=True` (the default) will instruct Zarr to write every chunk without checking for emptiness.

The following example illustrates the effect of the `write_empty_chunks` flag on the time required to write an array with different values.:

```
>>> import zarr
>>> import numpy as np
>>> import time
>>> from tempfile import TemporaryDirectory
>>> def timed_write(write_empty_chunks):
...     """
...     Measure the time required and number of objects created when writing
```

(continues on next page)

(continued from previous page)

```
...     to a Zarr array with random ints or fill value.
...     """
...     chunks = (8192,)
...     shape = (chunks[0] * 1024,)
...     data = np.random.randint(0, 255, shape)
...     dtype = 'uint8'
...
...     with TemporaryDirectory() as store:
...         arr = zarr.open(store,
...                         shape=shape,
...                         chunks=chunks,
...                         dtype=dtype,
...                         write_empty_chunks=write_empty_chunks,
...                         fill_value=0,
...                         mode='w')
...         # initialize all chunks
...         arr[:] = 100
...         result = []
...         for value in (data, arr.fill_value):
...             start = time.time()
...             arr[:] = value
...             elapsed = time.time() - start
...             result.append((elapsed, arr.nchunks_initialized))
...
...         return result
>>> for write_empty_chunks in (True, False):
...     full, empty = timed_write(write_empty_chunks)
...     print(f'\nwrite_empty_chunks={write_empty_chunks}:\n\tRandom Data: {full[0]:.4f}
↪s, {full[1]} objects stored\n\t Empty Data: {empty[0]:.4f}s, {empty[1]} objects stored\
↪n')

write_empty_chunks=True:
        Random Data: 0.1252s, 1024 objects stored
        Empty Data: 0.1060s, 1024 objects stored


write_empty_chunks=False:
        Random Data: 0.1359s, 1024 objects stored
        Empty Data: 0.0301s, 0 objects stored
```

In this example, writing random data is slightly slower with `write_empty_chunks=True`, but writing empty data is substantially faster and generates far fewer objects in storage.

2.15.4 Changing chunk shapes (rechunking)

Sometimes you are not free to choose the initial chunking of your input data, or you might have data saved with chunking which is not optimal for the analysis you have planned. In such cases it can be advantageous to re-chunk the data. For small datasets, or when the mismatch between input and output chunks is small such that only a few chunks of the input dataset need to be read to create each chunk in the output array, it is sufficient to simply copy the data to a new array with the desired chunking, e.g.

```
>>> a = zarr.zeros((10000, 10000), chunks=(100,100), dtype='uint16', store='a.zarr')
>>> b = zarr.array(a, chunks=(100, 200), store='b.zarr')
```

If the chunk shapes mismatch, however, a simple copy can lead to non-optimal data access patterns and incur a substantial performance hit when using file based stores. One of the most pathological examples is switching from column-based chunking to row-based chunking e.g.

```
>>> a = zarr.zeros((10000,10000), chunks=(10000, 1), dtype='uint16', store='a.zarr')
>>> b = zarr.array(a, chunks=(1,10000), store='b.zarr')
```

which will require every chunk in the input data set to be repeatedly read when creating each output chunk. If the entire array will fit within memory, this is simply resolved by forcing the entire input array into memory as a numpy array before converting back to zarr with the desired chunking.

```
>>> a = zarr.zeros((10000,10000), chunks=(10000, 1), dtype='uint16', store='a.zarr')
>>> b = a[...]
>>> c = zarr.array(b, chunks=(1,10000), store='c.zarr')
```

For data sets which have mismatched chunks and which do not fit in memory, a more sophisticated approach to rechunking, such as offered by the rechunker package and discussed here may offer a substantial improvement in performance.

2.16 Parallel computing and synchronization

Zarr arrays have been designed for use as the source or sink for data in parallel computations. By data source we mean that multiple concurrent read operations may occur. By data sink we mean that multiple concurrent write operations may occur, with each writer updating a different region of the array. Zarr arrays have **not** been designed for situations where multiple readers and writers are concurrently operating on the same array.

Both multi-threaded and multi-process parallelism are possible. The bottleneck for most storage and retrieval operations is compression/decompression, and the Python global interpreter lock (GIL) is released wherever possible during these operations, so Zarr will generally not block other Python threads from running.

When using a Zarr array as a data sink, some synchronization (locking) may be required to avoid data loss, depending on how data are being updated. If each worker in a parallel computation is writing to a separate region of the array, and if region boundaries are perfectly aligned with chunk boundaries, then no synchronization is required. However, if region and chunk boundaries are not perfectly aligned, then synchronization is required to avoid two workers attempting to modify the same chunk at the same time, which could result in data loss.

To give a simple example, consider a 1-dimensional array of length 60, `z`, divided into three chunks of 20 elements each. If three workers are running and each attempts to write to a 20 element region (i.e., `z[0:20]`, `z[20:40]` and `z[40:60]`) then each worker will be writing to a separate chunk and no synchronization is required. However, if two workers are running and each attempts to write to a 30 element region (i.e., `z[0:30]` and `z[30:60]`) then it is possible both workers will attempt to modify the middle chunk at the same time, and synchronization is required to prevent data loss.

Zarr provides support for chunk-level synchronization. E.g., create an array with thread synchronization:

```
>>> z = zarr.zeros((10000, 10000), chunks=(1000, 1000), dtype='i4',
...                synchronizer=zarr.ThreadSynchronizer())
>>> z
<zarr.core.Array (10000, 10000) int32>
```

This array is safe to read or write within a multi-threaded program.

Zarr also provides support for process synchronization via file locking, provided that all processes have access to a shared file system, and provided that the underlying file system supports file locking (which is not the case for some networked file systems). E.g.:

```
>>> synchronizer = zarr.ProcessSynchronizer('data/example.sync')
>>> z = zarr.open_array('data/example', mode='w', shape=(10000, 10000),
...                     chunks=(1000, 1000), dtype='i4',
...                     synchronizer=synchronizer)
>>> z
<zarr.core.Array (10000, 10000) int32>
```

This array is safe to read or write from multiple processes.

When using multiple processes to parallelize reads or writes on arrays using the Blosc compression library, it may be necessary to set `numcodecs.blosc.use_threads = False`, as otherwise Blosc may share incorrect global state amongst processes causing programs to hang. See also the section on *Configuring Blosc* below.

Please note that support for parallel computing is an area of ongoing research and development. If you are using Zarr for parallel computing, we welcome feedback, experience, discussion, ideas and advice, particularly about issues related to data integrity and performance.

2.17 Pickle support

Zarr arrays and groups can be pickled, as long as the underlying store object can be pickled. Instances of any of the storage classes provided in the *zarr.storage* module can be pickled, as can the built-in `dict` class which can also be used for storage.

Note that if an array or group is backed by an in-memory store like a `dict` or *zarr.storage.MemoryStore*, then when it is pickled all of the store data will be included in the pickled data. However, if an array or group is backed by a persistent store like a *zarr.storage.DirectoryStore*, *zarr.storage.ZipStore* or *zarr.storage.DBMStore* then the store data **are not** pickled. The only thing that is pickled is the necessary parameters to allow the store to re-open any underlying files or databases upon being unpickled.

E.g., pickle/unpickle an in-memory array:

```
>>> import pickle
>>> z1 = zarr.array(np.arange(100000))
>>> s = pickle.dumps(z1)
>>> len(s) > 5000  # relatively large because data have been pickled
True
>>> z2 = pickle.loads(s)
>>> z1 == z2
True
>>> np.all(z1[:] == z2[:])
True
```

E.g., pickle/unpickle an array stored on disk:

```
>>> z3 = zarr.open('data/walnuts.zarr', mode='w', shape=100000, dtype='i8')
>>> z3[:] = np.arange(100000)
>>> s = pickle.dumps(z3)
>>> len(s) < 200  # small because no data have been pickled
True
>>> z4 = pickle.loads(s)
>>> z3 == z4
True
>>> np.all(z3[:] == z4[:])
True
```

2.18 Datetimes and timedeltas

NumPy's `datetime64` ('M8') and `timedelta64` ('m8') dtypes are supported for Zarr arrays, as long as the units are specified. E.g.:

```
>>> z = zarr.array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='M8[D]')
>>> z
<zarr.core.Array (3,) datetime64[D]>
>>> z[:]
array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64[D]')
>>> z[0]
numpy.datetime64('2007-07-13')
>>> z[0] = '1999-12-31'
>>> z[:]
array(['1999-12-31', '2006-01-13', '2010-08-13'], dtype='datetime64[D]')
```

2.19 Usage tips

2.19.1 Copying large arrays

Data can be copied between large arrays without needing much memory, e.g.:

```
>>> z1 = zarr.empty((10000, 10000), chunks=(1000, 1000), dtype='i4')
>>> z1[:] = 42
>>> z2 = zarr.empty_like(z1)
>>> z2[:] = z1
```

Internally the example above works chunk-by-chunk, extracting only the data from `z1` required to fill each chunk in `z2`. The source of the data (`z1`) could equally be an h5py Dataset.

2.19.2 Configuring Blosc

The Blosc compressor is able to use multiple threads internally to accelerate compression and decompression. By default, Blosc uses up to 8 internal threads. The number of Blosc threads can be changed to increase or decrease this number, e.g.:

```
>>> from numcodecs import blosc
>>> blosc.set_nthreads(2)
8
```

When a Zarr array is being used within a multi-threaded program, Zarr automatically switches to using Blosc in a single-threaded "contextual" mode. This is generally better as it allows multiple program threads to use Blosc simultaneously and prevents CPU thrashing from too many active threads. If you want to manually override this behaviour, set the value of the `blosc.use_threads` variable to `True` (Blosc always uses multiple internal threads) or `False` (Blosc always runs in single-threaded contextual mode). To re-enable automatic switching, set `blosc.use_threads` to `None`.

Please note that if Zarr is being used within a multi-process program, Blosc may not be safe to use in multi-threaded mode and may cause the program to hang. If using Blosc in a multi-process program then it is recommended to set `blosc.use_threads = False`.

CHAPTER

THREE

API REFERENCE

3.1 Array creation (zarr.creation)

zarr.creation.**create**(*shape: int | Tuple[int, ...], chunks: int | Tuple[int, ...] | bool = True, dtype: dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any] = None, compressor='default', fill_value: int | None = 0, order: Literal['C', 'F'] = 'C', store: str | MutableMapping | None = None, synchronizer: Synchronizer | None = None, overwrite: bool = False, path: str | bytes | None = None, chunk_store: MutableMapping | None = None, filters: Sequence[Codec] | None = None, cache_metadata: bool = True, cache_attrs: bool = True, read_only: bool = False, object_codec: Codec | None = None, dimension_separator: Literal['.', '/'] | None = None, write_empty_chunks: bool = True, *, zarr_version: Literal[2, 3] | None = None, meta_array: MetaArray | None = None, storage_transformers: Sequence[StorageTransformer] = (), **kwargs*)

Create an array.

Parameters

shape
[int or tuple of ints] Array shape.

chunks
[int or tuple of ints, optional] Chunk shape. If True, will be guessed from *shape* and *dtype*. If False, will be set to *shape*, i.e., single chunk for the whole array. If an int, the chunk size in each dimension will be given by the value of *chunks*. Default is True.

dtype
[string or dtype, optional] NumPy dtype.

compressor
[Codec, optional] Primary compressor.

fill_value
[object] Default value to use for uninitialized portions of the array.

order
[{'C', 'F'}, optional] Memory layout to be used within each chunk.

store
[MutableMapping or string] Store or path to directory in file system or name of zip file.

synchronizer
[object, optional] Array synchronizer.

overwrite
[bool, optional] If True, delete all pre-existing data in *store* at *path* before creating the array.

path
[string, optional] Path under which array is stored.

chunk_store
[MutableMapping, optional] Separate storage for chunks. If not provided, *store* will be used for storage of both chunks and metadata.

filters
[sequence of Codecs, optional] Sequence of filters to use to encode chunk data prior to compression.

cache_metadata
[bool, optional] If True, array configuration metadata will be cached for the lifetime of the object. If False, array metadata will be reloaded prior to all data access and modification operations (may incur overhead depending on storage and data access pattern).

cache_attrs
[bool, optional] If True (default), user attributes will be cached for attribute read operations. If False, user attributes are reloaded from the store prior to all attribute read operations.

read_only
[bool, optional] True if array should be protected against modification.

object_codec
[Codec, optional] A codec to encode object arrays, only needed if dtype=object.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

New in version 2.8.

write_empty_chunks
[bool, optional] If True (default), all chunks will be stored regardless of their contents. If False, each chunk is compared to the array's fill value prior to storing. If a chunk is uniformly equal to the fill value, then that chunk is not be stored, and the store entry for that chunk's key is deleted. This setting enables sparser storage, as only chunks with non-fill-value data are stored, at the expense of overhead associated with checking the data of each chunk.

New in version 2.11.

storage_transformers
[sequence of StorageTransformers, optional] Setting storage transformers, changes the storage structure and behaviour of data coming from the underlying store. The transformers are applied in the order of the given sequence. Supplying an empty sequence is the same as omitting the argument or setting it to None. May only be set when using zarr_version 3.

New in version 2.13.

zarr_version
[{None, 2, 3}, optional] The zarr protocol version of the created array. If None, it will be inferred from `store` or `chunk_store` if they are provided, otherwise defaulting to 2.

New in version 2.12.

meta_array
[array-like, optional] An array instance to use for determining arrays to create and return to users. Use *numpy.empty(())* by default.

New in version 2.13.

Returns

z
[zarr.core.Array]

Examples

Create an array with default settings:

```
>>> import zarr
>>> z = zarr.create((10000, 10000), chunks=(1000, 1000))
>>> z
<zarr.core.Array (10000, 10000) float64>
```

Create an array with different some different configuration options:

```
>>> from numcodecs import Blosc
>>> compressor = Blosc(cname='zstd', clevel=1, shuffle=Blosc.BITSHUFFLE)
>>> z = zarr.create((10000, 10000), chunks=(1000, 1000), dtype='i1', order='F',
...                 compressor=compressor)
>>> z
<zarr.core.Array (10000, 10000) int8>
```

To create an array with object dtype requires a filter that can handle Python object encoding, e.g., *MsgPack* or *Pickle* from *numcodecs*:

```
>>> from numcodecs import MsgPack
>>> z = zarr.create((10000, 10000), chunks=(1000, 1000), dtype=object,
...                 object_codec=MsgPack())
>>> z
<zarr.core.Array (10000, 10000) object>
```

Example with some filters, and also storing chunks separately from metadata:

```
>>> from numcodecs import Quantize, Adler32
>>> store, chunk_store = dict(), dict()
>>> z = zarr.create((10000, 10000), chunks=(1000, 1000), dtype='f8',
...                 filters=[Quantize(digits=2, dtype='f8'), Adler32()],
...                 store=store, chunk_store=chunk_store)
>>> z
<zarr.core.Array (10000, 10000) float64>
```

zarr.creation.**empty**(*shape*, ***kwargs*)

Create an empty array.

For parameter definitions see *zarr.creation.create()*.

Notes

The contents of an empty Zarr array are not defined. On attempting to retrieve data from an empty Zarr array, any values may be returned, and these are not guaranteed to be stable from one access to the next.

zarr.creation.**zeros**(*shape*, ***kwargs*)

Create an array, with zero being used as the default value for uninitialized portions of the array.

For parameter definitions see *zarr.creation.create()*.

Examples

```
>>> import zarr
>>> z = zarr.zeros((10000, 10000), chunks=(1000, 1000))
>>> z
<zarr.core.Array (10000, 10000) float64>
>>> z[:2, :2]
array([[0., 0.],
       [0., 0.]])
```

zarr.creation.**ones**(*shape*, ***kwargs*)

Create an array, with one being used as the default value for uninitialized portions of the array.

For parameter definitions see *zarr.creation.create()*.

Examples

```
>>> import zarr
>>> z = zarr.ones((10000, 10000), chunks=(1000, 1000))
>>> z
<zarr.core.Array (10000, 10000) float64>
>>> z[:2, :2]
array([[1., 1.],
       [1., 1.]])
```

zarr.creation.**full**(*shape*, *fill_value*, ***kwargs*)

Create an array, with *fill_value* being used as the default value for uninitialized portions of the array.

For parameter definitions see *zarr.creation.create()*.

Examples

```
>>> import zarr
>>> z = zarr.full((10000, 10000), chunks=(1000, 1000), fill_value=42)
>>> z
<zarr.core.Array (10000, 10000) float64>
>>> z[:2, :2]
array([[42., 42.],
       [42., 42.]])
```

zarr.creation.**array**(*data*, ***kwargs*)

Create an array filled with *data*.

The *data* argument should be a NumPy array or array-like object. For other parameter definitions see *zarr.creation.create()*.

Examples

```
>>> import numpy as np
>>> import zarr
>>> a = np.arange(100000000).reshape(10000, 10000)
>>> z = zarr.array(a, chunks=(1000, 1000))
>>> z
<zarr.core.Array (10000, 10000) int64>
```

zarr.creation.**open_array**(*store=None*, *mode='a'*, *shape=None*, *chunks=True*, *dtype=None*, *compressor='default'*, *fill_value=0*, *order='C'*, *synchronizer=None*, *filters=None*, *cache_metadata=True*, *cache_attrs=True*, *path=None*, *object_codec=None*, *chunk_store=None*, *storage_options=None*, *partial_decompress=False*, *write_empty_chunks=True*, ***, *zarr_version=None*, *dimension_separator: Literal['.', '/'] | None = None*, *meta_array=None*, ***kwargs*)

Open an array using file-mode-like semantics.

Parameters

store
[MutableMapping or string, optional] Store or path to directory in file system or name of zip file.

mode
[{'r', 'r+', 'a', 'w', 'w-'}, optional] Persistence mode: 'r' means read only (must exist); 'r+' means read/write (must exist); 'a' means read/write (create if doesn't exist); 'w' means create (overwrite if exists); 'w-' means create (fail if exists).

shape
[int or tuple of ints, optional] Array shape.

chunks
[int or tuple of ints, optional] Chunk shape. If True, will be guessed from *shape* and *dtype*. If False, will be set to *shape*, i.e., single chunk for the whole array. If an int, the chunk size in each dimension will be given by the value of *chunks*. Default is True.

dtype
[string or dtype, optional] NumPy dtype.

compressor
[Codec, optional] Primary compressor.

fill_value
[object, optional] Default value to use for uninitialized portions of the array.

order
[{'C', 'F'}, optional] Memory layout to be used within each chunk.

synchronizer
[object, optional] Array synchronizer.

filters
[sequence, optional] Sequence of filters to use to encode chunk data prior to compression.

cache_metadata
[bool, optional] If True, array configuration metadata will be cached for the lifetime of the object. If False, array metadata will be reloaded prior to all data access and modification operations (may incur overhead depending on storage and data access pattern).

cache_attrs
[bool, optional] If True (default), user attributes will be cached for attribute read operations. If False, user attributes are reloaded from the store prior to all attribute read operations.

path
[string, optional] Array path within store.

object_codec
[Codec, optional] A codec to encode object arrays, only needed if dtype=object.

chunk_store
[MutableMapping or string, optional] Store or path to directory in file system or name of zip file.

storage_options
[dict] If using an fsspec URL to create the store, these will be passed to the backend implementation. Ignored otherwise.

partial_decompress
[bool, optional] If True and while the chunk_store is a FSStore and the compression used is Blosc, when getting data from the array chunks will be partially read and decompressed when possible.

write_empty_chunks
[bool, optional] If True (default), all chunks will be stored regardless of their contents. If False, each chunk is compared to the array's fill value prior to storing. If a chunk is uniformly equal to the fill value, then that chunk is not be stored, and the store entry for that chunk's key is deleted. This setting enables sparser storage, as only chunks with non-fill-value data are stored, at the expense of overhead associated with checking the data of each chunk.

New in version 2.11.

zarr_version
[{None, 2, 3}, optional] The zarr protocol version of the array to be opened. If None, it will be inferred from `store` or `chunk_store` if they are provided, otherwise defaulting to 2.

dimension_separator
[{None, '.', '/'}, optional] Can be used to specify whether the array is in a flat ('.') or nested ('/') format. If None, the appropriate value will be read from *store* when present. Otherwise, defaults to '.' when `zarr_version == 2` and `/` otherwise.

meta_array
[array-like, optional] An array instance to use for determining arrays to create and return to users. Use *numpy.empty(())* by default.

New in version 2.15.

Returns

z
[zarr.core.Array]

Notes

There is no need to close an array. Data are automatically flushed to the file system.

Examples

```
>>> import numpy as np
>>> import zarr
>>> z1 = zarr.open_array('data/example.zarr', mode='w', shape=(10000, 10000),
...                      chunks=(1000, 1000), fill_value=0)
>>> z1[:] = np.arange(100000000).reshape(10000, 10000)
>>> z1
<zarr.core.Array (10000, 10000) float64>
>>> z2 = zarr.open_array('data/example.zarr', mode='r')
>>> z2
<zarr.core.Array (10000, 10000) float64 read-only>
>>> np.all(z1[:] == z2[:])
True
```

zarr.creation.**empty_like**(*a*, ***kwargs*)

Create an empty array like *a*.

zarr.creation.**zeros_like**(*a*, ***kwargs*)

Create an array of zeros like *a*.

zarr.creation.**ones_like**(*a*, ***kwargs*)

Create an array of ones like *a*.

zarr.creation.**full_like**(*a*, ***kwargs*)

Create a filled array like *a*.

zarr.creation.**open_like**(*a*, *path*, ***kwargs*)

Open a persistent array like *a*.

3.2 The Array class (zarr.core)

3.2.1 Classes

Array(store[, path, read_only, chunk_store, ...])	Instantiate an array from an initialized store.

Array

class zarr.core.**Array**(*store: Any*, *path=None*, *read_only=False*, *chunk_store=None*, *synchronizer=None*, *cache_metadata=True*, *cache_attrs=True*, *partial_decompress=False*, *write_empty_chunks=True*, *zarr_version=None*, *meta_array=None*)

Bases: object

Instantiate an array from an initialized store.

Parameters

store
[MutableMapping] Array store, already initialized.

path
[string, optional] Storage path.

read_only
[bool, optional] True if array should be protected against modification.

chunk_store
[MutableMapping, optional] Separate storage for chunks. If not provided, *store* will be used for storage of both chunks and metadata.

synchronizer
[object, optional] Array synchronizer.

cache_metadata
[bool, optional] If True (default), array configuration metadata will be cached for the lifetime of the object. If False, array metadata will be reloaded prior to all data access and modification operations (may incur overhead depending on storage and data access pattern).

cache_attrs
[bool, optional] If True (default), user attributes will be cached for attribute read operations. If False, user attributes are reloaded from the store prior to all attribute read operations.

partial_decompress
[bool, optional] If True and while the chunk_store is a FSStore and the compression used is Blosc, when getting data from the array chunks will be partially read and decompressed when possible.

New in version 2.7.

write_empty_chunks
[bool, optional] If True, all chunks will be stored regardless of their contents. If False (default), each chunk is compared to the array's fill value prior to storing. If a chunk is uniformly equal to the fill value, then that chunk is not be stored, and the store entry for that chunk's key is deleted. This setting enables sparser storage, as only chunks with non-fill-value data are stored, at the expense of overhead associated with checking the data of each chunk.

New in version 2.11.

meta_array
[array-like, optional] An array instance to use for determining arrays to create and return to users. Use *numpy.empty(())* by default.

New in version 2.13.

Attributes Summary

attrs	A MutableMapping containing user-defined attributes.
basename	Final component of name.
blocks	Shortcut for blocked chunked indexing, see *get_block_selection()* and *set_block_selection()* for documentation and examples.

continues on next page

Table 1 – continued from previous page

cdata_shape	A tuple of integers describing the number of chunks along each dimension of the array.
chunk_store	A MutableMapping providing the underlying storage for array chunks.
chunks	A tuple of integers describing the length of each dimension of a chunk of the array.
compressor	Primary compression codec.
dtype	The NumPy data type.
fill_value	A value used for uninitialized portions of the array.
filters	One or more codecs used to transform data prior to compression.
info	Report some diagnostic information about the array.
initialized	The number of chunks that have been initialized with some data.
is_view	A boolean, True if this array is a view on another array.
itemsize	The size in bytes of each item in the array.
meta_array	An array-like instance to use for determining arrays to create and return to users.
name	Array name following h5py convention.
nbytes	The total number of bytes that would be required to store the array without compression.
nbytes_stored	The total number of stored bytes of data for the array.
nchunks	Total number of chunks.
nchunks_initialized	The number of chunks that have been initialized with some data.
ndim	Number of dimensions.
oindex	Shortcut for orthogonal (outer) indexing, see *get_orthogonal_selection()* and *set_orthogonal_selection()* for documentation and examples.
order	A string indicating the order in which bytes are arranged within chunks of the array.
path	Storage path.
read_only	A boolean, True if modification operations are not permitted.
shape	A tuple of integers describing the length of each dimension of the array.
size	The total number of elements in the array.
store	A MutableMapping providing the underlying storage for the array.
synchronizer	Object used to synchronize write access to the array.
vindex	Shortcut for vectorized (inner) indexing, see *get_coordinate_selection()*, *set_coordinate_selection()*, *get_mask_selection()* and *set_mask_selection()* for documentation and examples.
write_empty_chunks	A Boolean, True if chunks composed of the array's fill value will be stored.

Methods Summary

append(data[, axis])	Append *data* to *axis*.
astype(dtype)	Returns a view that does on the fly type conversion of the underlying data.
digest([hashname])	Compute a checksum for the data.
get_basic_selection([selection, out, fields])	Retrieve data for an item or region of the array.
get_block_selection(selection[, out, fields])	Retrieve a selection of individual chunk blocks, by providing the indices (coordinates) for each chunk block.
get_coordinate_selection(selection[, out, ...])	Retrieve a selection of individual items, by providing the indices (coordinates) for each selected item.
get_mask_selection(selection[, out, fields])	Retrieve a selection of individual items, by providing a Boolean array of the same shape as the array against which the selection is being made, where True values indicate a selected item.
get_orthogonal_selection(selection[, out, ...])	Retrieve data by making a selection for each dimension of the array.
hexdigest([hashname])	Compute a checksum for the data.
info_items()	
islice([start, end])	Yield a generator for iterating over the entire or parts of the array.
resize(*args)	Change the shape of the array by growing or shrinking one or more dimensions.
set_basic_selection(selection, value[, fields])	Modify data for an item or region of the array.
set_block_selection(selection, value[, fields])	Modify a selection of individual blocks, by providing the chunk indices (coordinates) for each block to be modified.
set_coordinate_selection(selection, value[, ...])	Modify a selection of individual items, by providing the indices (coordinates) for each item to be modified.
set_mask_selection(selection, value[, fields])	Modify a selection of individual items, by providing a Boolean array of the same shape as the array against which the selection is being made, where True values indicate a selected item.
set_orthogonal_selection(selection, value[, ...])	Modify data via a selection for each dimension of the array.
view([shape, chunks, dtype, fill_value, ...])	Return an array sharing the same data.

Attributes Documentation

attrs

A MutableMapping containing user-defined attributes. Note that attribute values must be JSON serializable.

basename

Final component of name.

blocks

Shortcut for blocked chunked indexing, see *get_block_selection()* and *set_block_selection()* for documentation and examples.

cdata_shape

A tuple of integers describing the number of chunks along each dimension of the array.

chunk_store

A MutableMapping providing the underlying storage for array chunks.

chunks

A tuple of integers describing the length of each dimension of a chunk of the array.

compressor

Primary compression codec.

dtype

The NumPy data type.

fill_value

A value used for uninitialized portions of the array.

filters

One or more codecs used to transform data prior to compression.

info

Report some diagnostic information about the array.

Examples

```
>>> import zarr
>>> z = zarr.zeros(1000000, chunks=100000, dtype='i4')
>>> z.info
Type               : zarr.core.Array
Data type          : int32
Shape              : (1000000,)
Chunk shape        : (100000,)
Order              : C
Read-only          : False
Compressor         : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type         : zarr.storage.KVStore
No. bytes          : 4000000 (3.8M)
No. bytes stored   : 320
Storage ratio      : 12500.0
Chunks initialized : 0/10
```

initialized

The number of chunks that have been initialized with some data.

is_view

A boolean, True if this array is a view on another array.

itemsize

The size in bytes of each item in the array.

meta_array

An array-like instance to use for determining arrays to create and return to users.

name

Array name following h5py convention.

nbytes

The total number of bytes that would be required to store the array without compression.

nbytes_stored

The total number of stored bytes of data for the array. This includes storage required for configuration metadata and user attributes.

nchunks

Total number of chunks.

nchunks_initialized

The number of chunks that have been initialized with some data.

ndim

Number of dimensions.

oindex

Shortcut for orthogonal (outer) indexing, see *get_orthogonal_selection()* and *set_orthogonal_selection()* for documentation and examples.

order

A string indicating the order in which bytes are arranged within chunks of the array.

path

Storage path.

read_only

A boolean, True if modification operations are not permitted.

shape

A tuple of integers describing the length of each dimension of the array.

size

The total number of elements in the array.

store

A MutableMapping providing the underlying storage for the array.

synchronizer

Object used to synchronize write access to the array.

vindex

Shortcut for vectorized (inner) indexing, see *get_coordinate_selection()*, *set_coordinate_selection()*, *get_mask_selection()* and *set_mask_selection()* for documentation and examples.

write_empty_chunks

A Boolean, True if chunks composed of the array's fill value will be stored. If False, such chunks will not be stored.

Methods Documentation

append(*data*, *axis=0*)

Append *data* to *axis*.

Parameters

data
[array-like] Data to be appended.

axis
[int] Axis along which to append.

Returns

new_shape
[tuple]

Notes

The size of all dimensions other than *axis* must match between this array and *data*.

Examples

```
>>> import numpy as np
>>> import zarr
>>> a = np.arange(10000000, dtype='i4').reshape(10000, 1000)
>>> z = zarr.array(a, chunks=(1000, 100))
>>> z.shape
(10000, 1000)
>>> z.append(a)
(20000, 1000)
>>> z.append(np.vstack([a, a]), axis=1)
(20000, 2000)
>>> z.shape
(20000, 2000)
```

astype(*dtype*)

Returns a view that does on the fly type conversion of the underlying data.

Parameters

dtype
[string or dtype] NumPy dtype.

See also:

Array.view

Notes

This method returns a new Array object which is a view on the same underlying chunk data. Modifying any data via the view is currently not permitted and will result in an error. This is an experimental feature and its behavior is subject to change in the future.

Examples

```
>>> import zarr
>>> import numpy as np
>>> data = np.arange(100, dtype=np.uint8)
>>> a = zarr.array(data, chunks=10)
>>> 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, 24, 25, 26, 27, 28, 29, 30, 31,
       32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
       48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
       64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
       80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
       96, 97, 98, 99], dtype=uint8)
>>> v = a.astype(np.float32)
>>> v.is_view
True
>>> v[:]
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.,  24.,  25.,  26.,  27.,  28.,  29.,
        30.,  31.,  32.,  33.,  34.,  35.,  36.,  37.,  38.,  39.,
        40.,  41.,  42.,  43.,  44.,  45.,  46.,  47.,  48.,  49.,
        50.,  51.,  52.,  53.,  54.,  55.,  56.,  57.,  58.,  59.,
        60.,  61.,  62.,  63.,  64.,  65.,  66.,  67.,  68.,  69.,
        70.,  71.,  72.,  73.,  74.,  75.,  76.,  77.,  78.,  79.,
        80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,  88.,  89.,
        90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,  99.],
      dtype=float32)
```

digest(*hashname='sha1'*)

Compute a checksum for the data. Default uses sha1 for speed.

Examples

```
>>> import binascii
>>> import zarr
>>> z = zarr.empty(shape=(10000, 10000), chunks=(1000, 1000))
>>> binascii.hexlify(z.digest())
b'041f90bc7a571452af4f850a8ca2c6cddfa8a1ac'
>>> z = zarr.zeros(shape=(10000, 10000), chunks=(1000, 1000))
>>> binascii.hexlify(z.digest())
b'7162d416d26a68063b66ed1f30e0a866e4abed60'
>>> z = zarr.zeros(shape=(10000, 10000), dtype="u1", chunks=(1000, 1000))
```

(continues on next page)

(continued from previous page)

```
>>> binascii.hexlify(z.digest())
b'cb387af37410ae5a3222e893cf3373e4e4f22816'
```

get_basic_selection(*selection=Ellipsis*, *out=None*, *fields=None*)

Retrieve data for an item or region of the array.

Parameters

selection
[tuple] A tuple specifying the requested item or region for each dimension of the array. May be any combination of int and/or slice for multidimensional arrays.

out
[ndarray, optional] If given, load the selected data directly into this array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to extract data for.

Returns

out
[ndarray] A NumPy array containing the data for the requested region.

See also:

set_basic_selection, *get_mask_selection*, *set_mask_selection*
get_coordinate_selection, *set_coordinate_selection*, *get_orthogonal_selection*
set_orthogonal_selection, *get_block_selection*, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Slices with step > 1 are supported, but slices with negative step are not.

Currently this method provides the implementation for accessing data via the square bracket notation (__getitem__). See `__getitem__()` for examples using the alternative notation.

Examples

Setup a 1-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.array(np.arange(100))
```

Retrieve a single item:

```
>>> z.get_basic_selection(5)
5
```

Retrieve a region via slicing:

```
>>> z.get_basic_selection(slice(5))
array([0, 1, 2, 3, 4])
>>> z.get_basic_selection(slice(-5, None))
array([95, 96, 97, 98, 99])
>>> z.get_basic_selection(slice(5, 10))
array([5, 6, 7, 8, 9])
>>> z.get_basic_selection(slice(5, 10, 2))
array([5, 7, 9])
>>> z.get_basic_selection(slice(None, None, 2))
array([  0,  2,  4, ..., 94, 96, 98])
```

Setup a 2-dimensional array:

```
>>> z = zarr.array(np.arange(100).reshape(10, 10))
```

Retrieve an item:

```
>>> z.get_basic_selection((2, 2))
22
```

Retrieve a region via slicing:

```
>>> z.get_basic_selection((slice(1, 3), slice(1, 3)))
array([[11, 12],
       [21, 22]])
>>> z.get_basic_selection((slice(1, 3), slice(None)))
array([[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])
>>> z.get_basic_selection((slice(None), slice(1, 3)))
array([[ 1,  2],
       [11, 12],
       [21, 22],
       [31, 32],
       [41, 42],
       [51, 52],
       [61, 62],
       [71, 72],
       [81, 82],
       [91, 92]])
>>> z.get_basic_selection((slice(0, 5, 2), slice(0, 5, 2)))
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
>>> z.get_basic_selection((slice(None, None, 2), slice(None, None, 2)))
array([[ 0,  2,  4,  6,  8],
       [20, 22, 24, 26, 28],
       [40, 42, 44, 46, 48],
       [60, 62, 64, 66, 68],
       [80, 82, 84, 86, 88]])
```

For arrays with a structured dtype, specific fields can be retrieved, e.g.:

```
>>> a = np.array([(b'aaa', 1, 4.2),
...               (b'bbb', 2, 8.4),
```

(continues on next page)

(continued from previous page)

```
...                (b'ccc', 3, 12.6)],
...               dtype=[('foo', 'S3'), ('bar', 'i4'), ('baz', 'f8')])
>>> z = zarr.array(a)
>>> z.get_basic_selection(slice(2), fields='foo')
array([b'aaa', b'bbb'],
      dtype='|S3')
```

get_block_selection(*selection*, *out=None*, *fields=None*)

Retrieve a selection of individual chunk blocks, by providing the indices (coordinates) for each chunk block.

Parameters

selection
[tuple] An integer (coordinate) or slice for each dimension of the array.

out
[ndarray, optional] If given, load the selected data directly into this array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to extract data for.

Returns

out
[ndarray] A NumPy array containing the data for the requested selection.

See also:

get_basic_selection, *set_basic_selection*, *get_mask_selection*, *set_mask_selection*
get_orthogonal_selection, *set_orthogonal_selection*, *get_coordinate_selection*
set_coordinate_selection, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Block indexing is a convenience indexing method to work on individual chunks with chunk index slicing. It has the same concept as Dask's *Array.blocks* indexing.

Slices are supported. However, only with a step size of one.

Block index arrays may be multidimensional to index multidimensional arrays. For example:

```
>>> z.blocks[0, 1:3]
array([[ 3,  4,  5,  6,  7,  8],
       [13, 14, 15, 16, 17, 18],
       [23, 24, 25, 26, 27, 28]])
```

Examples

Setup a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.array(np.arange(100).reshape(10, 10), chunks=(3, 3))
```

Retrieve items by specifying their block coordinates:

```
>>> z.get_block_selection((1, slice(None)))
array([[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

Which is equivalent to:

```
>>> z[3:6, :]
array([[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

For convenience, the block selection functionality is also available via the *blocks* property, e.g.:

```
>>> z.blocks[1]
array([[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

get_coordinate_selection(*selection*, *out=None*, *fields=None*)

Retrieve a selection of individual items, by providing the indices (coordinates) for each selected item.

Parameters

selection
[tuple] An integer (coordinate) array for each dimension of the array.

out
[ndarray, optional] If given, load the selected data directly into this array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to extract data for.

Returns

out
[ndarray] A NumPy array containing the data for the requested selection.

See also:

get_basic_selection, *set_basic_selection*, *get_mask_selection*, *set_mask_selection*
get_orthogonal_selection, *set_orthogonal_selection*, *set_coordinate_selection*
get_block_selection, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Coordinate indexing is also known as point selection, and is a form of vectorized or inner indexing.

Slices are not supported. Coordinate arrays must be provided for all dimensions of the array.

Coordinate arrays may be multidimensional, in which case the output array will also be multidimensional. Coordinate arrays are broadcast against each other before being applied. The shape of the output will be the same as the shape of each coordinate array after broadcasting.

Examples

Setup a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.array(np.arange(100).reshape(10, 10))
```

Retrieve items by specifying their coordinates:

```
>>> z.get_coordinate_selection(([1, 4], [1, 4]))
array([11, 44])
```

For convenience, the coordinate selection functionality is also available via the *vindex* property, e.g.:

```
>>> z.vindex[[1, 4], [1, 4]]
array([11, 44])
```

get_mask_selection(*selection*, *out=None*, *fields=None*)

Retrieve a selection of individual items, by providing a Boolean array of the same shape as the array against which the selection is being made, where True values indicate a selected item.

Parameters

selection
[ndarray, bool] A Boolean array of the same shape as the array against which the selection is being made.

out
[ndarray, optional] If given, load the selected data directly into this array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to extract data for.

Returns

out
[ndarray] A NumPy array containing the data for the requested selection.

See also:

get_basic_selection, *set_basic_selection*, *set_mask_selection*
get_orthogonal_selection, *set_orthogonal_selection*, *get_coordinate_selection*
set_coordinate_selection, *get_block_selection*, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Mask indexing is a form of vectorized or inner indexing, and is equivalent to coordinate indexing. Internally the mask array is converted to coordinate arrays by calling *np.nonzero*.

Examples

Setup a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.array(np.arange(100).reshape(10, 10))
```

Retrieve items by specifying a mask:

```
>>> sel = np.zeros_like(z, dtype=bool)
>>> sel[1, 1] = True
>>> sel[4, 4] = True
>>> z.get_mask_selection(sel)
array([11, 44])
```

For convenience, the mask selection functionality is also available via the *vindex* property, e.g.:

```
>>> z.vindex[sel]
array([11, 44])
```

get_orthogonal_selection(*selection*, *out=None*, *fields=None*)

Retrieve data by making a selection for each dimension of the array. For example, if an array has 2 dimensions, allows selecting specific rows and/or columns. The selection for each dimension can be either an integer (indexing a single item), a slice, an array of integers, or a Boolean array where True values indicate a selection.

Parameters

selection
[tuple] A selection for each dimension of the array. May be any combination of int, slice, integer array or Boolean array.

out
[ndarray, optional] If given, load the selected data directly into this array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to extract data for.

Returns

out
[ndarray] A NumPy array containing the data for the requested selection.

See also:

get_basic_selection, *set_basic_selection*, *get_mask_selection*, *set_mask_selection*
get_coordinate_selection, *set_coordinate_selection*, *set_orthogonal_selection*
get_block_selection, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Orthogonal indexing is also known as outer indexing.

Slices with step > 1 are supported, but slices with negative step are not.

Examples

Setup a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.array(np.arange(100).reshape(10, 10))
```

Retrieve rows and columns via any combination of int, slice, integer array and/or Boolean array:

```
>>> z.get_orthogonal_selection(([1, 4], slice(None)))
array([[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
>>> z.get_orthogonal_selection((slice(None), [1, 4]))
array([[ 1,  4],
       [11, 14],
       [21, 24],
       [31, 34],
       [41, 44],
       [51, 54],
       [61, 64],
       [71, 74],
       [81, 84],
       [91, 94]])
>>> z.get_orthogonal_selection(([1, 4], [1, 4]))
array([[11, 14],
       [41, 44]])
>>> sel = np.zeros(z.shape[0], dtype=bool)
>>> sel[1] = True
>>> sel[4] = True
>>> z.get_orthogonal_selection((sel, sel))
array([[11, 14],
       [41, 44]])
```

For convenience, the orthogonal selection functionality is also available via the *oindex* property, e.g.:

```
>>> z.oindex[[1, 4], :]
array([[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
>>> z.oindex[:, [1, 4]]
array([[ 1,  4],
       [11, 14],
       [21, 24],
       [31, 34],
       [41, 44],
       [51, 54],
       [61, 64],
       [71, 74],
```

(continues on next page)

(continued from previous page)

```
       [81, 84],
       [91, 94]])
>>> z.oindex[[1, 4], [1, 4]]
array([[11, 14],
       [41, 44]])
>>> sel = np.zeros(z.shape[0], dtype=bool)
>>> sel[1] = True
>>> sel[4] = True
>>> z.oindex[sel, sel]
array([[11, 14],
       [41, 44]])
```

hexdigest(*hashname='sha1'*)

Compute a checksum for the data. Default uses sha1 for speed.

Examples

```
>>> import zarr
>>> z = zarr.empty(shape=(10000, 10000), chunks=(1000, 1000))
>>> z.hexdigest()
'041f90bc7a571452af4f850a8ca2c6cddfa8a1ac'
>>> z = zarr.zeros(shape=(10000, 10000), chunks=(1000, 1000))
>>> z.hexdigest()
'7162d416d26a68063b66ed1f30e0a866e4abed60'
>>> z = zarr.zeros(shape=(10000, 10000), dtype="u1", chunks=(1000, 1000))
>>> z.hexdigest()
'cb387af37410ae5a3222e893cf3373e4e4f22816'
```

info_items()

islice(*start=None*, *end=None*)

Yield a generator for iterating over the entire or parts of the array. Uses a cache so chunks only have to be decompressed once.

Parameters

start
[int, optional] Start index for the generator to start at. Defaults to 0.

end
[int, optional] End index for the generator to stop at. Defaults to self.shape[0].

Yields

out
[generator] A generator that can be used to iterate over the requested region the array.

Examples

Setup a 1-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.array(np.arange(100))
```

Iterate over part of the array:

```
>>> for value in z.islice(25, 30): value;
25
26
27
28
29
```

resize(**args*)

Change the shape of the array by growing or shrinking one or more dimensions.

Notes

When resizing an array, the data are not rearranged in any way.

If one or more dimensions are shrunk, any chunks falling outside the new array shape will be deleted from the underlying store. However, it is noteworthy that the chunks partially falling inside the new array (i.e. boundary chunks) will remain intact, and therefore, the data falling outside the new array but inside the boundary chunks would be restored by a subsequent resize operation that grows the array size.

Examples

```
>>> import zarr
>>> z = zarr.zeros(shape=(10000, 10000), chunks=(1000, 1000))
>>> z.shape
(10000, 10000)
>>> z.resize(20000, 10000)
>>> z.shape
(20000, 10000)
>>> z.resize(30000, 1000)
>>> z.shape
(30000, 1000)
```

set_basic_selection(*selection*, *value*, *fields=None*)

Modify data for an item or region of the array.

Parameters

selection
[tuple] An integer index or slice or tuple of int/slice specifying the requested region for each dimension of the array.

value
[scalar or array-like] Value to be stored into the array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to set data for.

See also:

get_basic_selection, *get_mask_selection*, *set_mask_selection*
get_coordinate_selection, *set_coordinate_selection*, *get_orthogonal_selection*
set_orthogonal_selection, *get_block_selection*, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

This method provides the underlying implementation for modifying data via square bracket notation, see `__setitem__()` for equivalent examples using the alternative notation.

Examples

Setup a 1-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.zeros(100, dtype=int)
```

Set all array elements to the same scalar value:

```
>>> z.set_basic_selection(..., 42)
>>> z[...]
array([42, 42, 42, ..., 42, 42, 42])
```

Set a portion of the array:

```
>>> z.set_basic_selection(slice(10), np.arange(10))
>>> z.set_basic_selection(slice(-10, None), np.arange(10)[::-1])
>>> z[...]
array([ 0, 1, 2, ..., 2, 1, 0])
```

Setup a 2-dimensional array:

```
>>> z = zarr.zeros((5, 5), dtype=int)
```

Set all array elements to the same scalar value:

```
>>> z.set_basic_selection(..., 42)
```

Set a portion of the array:

```
>>> z.set_basic_selection((0, slice(None)), np.arange(z.shape[1]))
>>> z.set_basic_selection((slice(None), 0), np.arange(z.shape[0]))
>>> z[...]
array([[ 0,  1,  2,  3,  4],
       [ 1, 42, 42, 42, 42],
       [ 2, 42, 42, 42, 42],
```

(continues on next page)

(continued from previous page)

```
       [ 3, 42, 42, 42, 42],
       [ 4, 42, 42, 42, 42]])
```

For arrays with a structured dtype, the *fields* parameter can be used to set data for a specific field, e.g.:

```
>>> a = np.array([(b'aaa', 1, 4.2),
...               (b'bbb', 2, 8.4),
...               (b'ccc', 3, 12.6)],
...              dtype=[('foo', 'S3'), ('bar', 'i4'), ('baz', 'f8')])
>>> z = zarr.array(a)
>>> z.set_basic_selection(slice(0, 2), b'zzz', fields='foo')
>>> z[:]
array([(b'zzz', 1,  4.2), (b'zzz', 2,  8.4), (b'ccc', 3, 12.6)],
      dtype=[('foo', 'S3'), ('bar', '<i4'), ('baz', '<f8')])
```

set_block_selection(*selection*, *value*, *fields=None*)

Modify a selection of individual blocks, by providing the chunk indices (coordinates) for each block to be modified.

Parameters

selection
[tuple] An integer (coordinate) or slice for each dimension of the array.

value
[scalar or array-like] Value to be stored into the array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to set data for.

See also:

get_basic_selection, *set_basic_selection*, *get_mask_selection*, *set_mask_selection*
get_orthogonal_selection, *set_orthogonal_selection*, *get_coordinate_selection*
get_block_selection, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Block indexing is a convenience indexing method to work on individual chunks with chunk index slicing. It has the same concept as Dask's *Array.blocks* indexing.

Slices are supported. However, only with a step size of one.

Examples

Set up a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.zeros((6, 6), dtype=int, chunks=2)
```

Set data for a selection of items:

```
>>> z.set_block_selection((1, 0), 1)
>>> z[...]
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
```

For convenience, this functionality is also available via the *blocks* property. E.g.:

```
>>> z.blocks[2, 1] = 4
>>> z[...]
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0],
       [0, 0, 4, 4, 0, 0],
       [0, 0, 4, 4, 0, 0]])

>>> z.blocks[:, 2] = 7
>>> z[...]
array([[0, 0, 0, 0, 7, 7],
       [0, 0, 0, 0, 7, 7],
       [1, 1, 0, 0, 7, 7],
       [1, 1, 0, 0, 7, 7],
       [0, 0, 4, 4, 7, 7],
       [0, 0, 4, 4, 7, 7]])
```

set_coordinate_selection(*selection*, *value*, *fields=None*)

Modify a selection of individual items, by providing the indices (coordinates) for each item to be modified.

Parameters

selection
[tuple] An integer (coordinate) array for each dimension of the array.

value
[scalar or array-like] Value to be stored into the array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to set data for.

See also:

get_basic_selection, *set_basic_selection*, *get_mask_selection*, *set_mask_selection*
get_orthogonal_selection, *set_orthogonal_selection*, *get_coordinate_selection*
get_block_selection, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Coordinate indexing is also known as point selection, and is a form of vectorized or inner indexing.

Slices are not supported. Coordinate arrays must be provided for all dimensions of the array.

Examples

Setup a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.zeros((5, 5), dtype=int)
```

Set data for a selection of items:

```
>>> z.set_coordinate_selection(([1, 4], [1, 4]), 1)
>>> z[...]
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1]])
```

For convenience, this functionality is also available via the *vindex* property. E.g.:

```
>>> z.vindex[[1, 4], [1, 4]] = 2
>>> z[...]
array([[0, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 2]])
```

set_mask_selection(*selection*, *value*, *fields=None*)

Modify a selection of individual items, by providing a Boolean array of the same shape as the array against which the selection is being made, where True values indicate a selected item.

Parameters

selection
[ndarray, bool] A Boolean array of the same shape as the array against which the selection is being made.

value
[scalar or array-like] Value to be stored into the array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to set data for.

See also:

get_basic_selection, *set_basic_selection*, *get_mask_selection*
get_orthogonal_selection, *set_orthogonal_selection*, *get_coordinate_selection*
set_coordinate_selection, *get_block_selection*, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Mask indexing is a form of vectorized or inner indexing, and is equivalent to coordinate indexing. Internally the mask array is converted to coordinate arrays by calling *np.nonzero*.

Examples

Setup a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.zeros((5, 5), dtype=int)
```

Set data for a selection of items:

```
>>> sel = np.zeros_like(z, dtype=bool)
>>> sel[1, 1] = True
>>> sel[4, 4] = True
>>> z.set_mask_selection(sel, 1)
>>> z[...]
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1]])
```

For convenience, this functionality is also available via the *vindex* property. E.g.:

```
>>> z.vindex[sel] = 2
>>> z[...]
array([[0, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 2]])
```

set_orthogonal_selection(*selection*, *value*, *fields=None*)

Modify data via a selection for each dimension of the array.

Parameters

selection
[tuple] A selection for each dimension of the array. May be any combination of int, slice, integer array or Boolean array.

value
[scalar or array-like] Value to be stored into the array.

fields
[str or sequence of str, optional] For arrays with a structured dtype, one or more fields can be specified to set data for.

See also:

get_basic_selection, *set_basic_selection*, *get_mask_selection*, *set_mask_selection*
get_coordinate_selection, *set_coordinate_selection*, *get_orthogonal_selection*
get_block_selection, *set_block_selection*
vindex, *oindex*, *blocks*, **__getitem__**, **__setitem__**

Notes

Orthogonal indexing is also known as outer indexing.

Slices with step > 1 are supported, but slices with negative step are not.

Examples

Setup a 2-dimensional array:

```
>>> import zarr
>>> import numpy as np
>>> z = zarr.zeros((5, 5), dtype=int)
```

Set data for a selection of rows:

```
>>> z.set_orthogonal_selection(([1, 4], slice(None)), 1)
>>> z[...]
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1]])
```

Set data for a selection of columns:

```
>>> z.set_orthogonal_selection((slice(None), [1, 4]), 2)
>>> z[...]
array([[0, 2, 0, 0, 2],
       [1, 2, 1, 1, 2],
       [0, 2, 0, 0, 2],
       [0, 2, 0, 0, 2],
       [1, 2, 1, 1, 2]])
```

Set data for a selection of rows and columns:

```
>>> z.set_orthogonal_selection(([1, 4], [1, 4]), 3)
>>> z[...]
array([[0, 2, 0, 0, 2],
       [1, 3, 1, 1, 3],
       [0, 2, 0, 0, 2],
       [0, 2, 0, 0, 2],
       [1, 3, 1, 1, 3]])
```

For convenience, this functionality is also available via the *oindex* property. E.g.:

```
>>> z.oindex[[1, 4], [1, 4]] = 4
>>> z[...]
array([[0, 2, 0, 0, 2],
       [1, 4, 1, 1, 4],
       [0, 2, 0, 0, 2],
       [0, 2, 0, 0, 2],
       [1, 4, 1, 1, 4]])
```

view(*shape=None*, *chunks=None*, *dtype=None*, *fill_value=None*, *filters=None*, *read_only=None*, *synchronizer=None*)

Return an array sharing the same data.

Parameters

shape
[int or tuple of ints] Array shape.

chunks
[int or tuple of ints, optional] Chunk shape.

dtype
[string or dtype, optional] NumPy dtype.

fill_value
[object] Default value to use for uninitialized portions of the array.

filters
[sequence, optional] Sequence of filters to use to encode chunk data prior to compression.

read_only
[bool, optional] True if array should be protected against modification.

synchronizer
[object, optional] Array synchronizer.

Notes

WARNING: This is an experimental feature and should be used with care. There are plenty of ways to generate errors and/or cause data corruption.

Examples

Bypass filters:

```
>>> import zarr
>>> import numpy as np
>>> np.random.seed(42)
>>> labels = ['female', 'male']
>>> data = np.random.choice(labels, size=10000)
>>> filters = [zarr.Categorize(labels=labels,
...                            dtype=data.dtype,
...                            astype='u1')]
>>> a = zarr.array(data, chunks=1000, filters=filters)
>>> a[:]
array(['female', 'male', 'female', ..., 'male', 'male', 'female'],
      dtype='<U6')
>>> v = a.view(dtype='u1', filters=[])
>>> v.is_view
True
>>> v[:]
array([1, 2, 1, ..., 2, 2, 1], dtype=uint8)
```

Views can be used to modify data:

```
>>> x = v[:]
>>> x.sort()
>>> v[:] = x
>>> v[:]
array([1, 1, 1, ..., 2, 2, 2], dtype=uint8)
>>> a[:]
array(['female', 'female', 'female', ..., 'male', 'male', 'male'],
      dtype='<U6')
```

View as a different dtype with the same item size:

```
>>> data = np.random.randint(0, 2, size=10000, dtype='u1')
>>> a = zarr.array(data, chunks=1000)
>>> a[:]
array([0, 0, 1, ..., 1, 0, 0], dtype=uint8)
>>> v = a.view(dtype=bool)
>>> v[:]
array([False, False,  True, ...,  True, False, False])
>>> np.all(a[:].view(dtype=bool) == v[:])
True
```

An array can be viewed with a dtype with a different item size, however some care is needed to adjust the shape and chunk shape so that chunk data is interpreted correctly:

```
>>> data = np.arange(10000, dtype='u2')
>>> a = zarr.array(data, chunks=1000)
>>> a[:10]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint16)
>>> v = a.view(dtype='u1', shape=20000, chunks=2000)
>>> v[:10]
```

(continues on next page)

(continued from previous page)

```
array([0, 0, 1, 0, 2, 0, 3, 0, 4, 0], dtype=uint8)
>>> np.all(a[:].view('u1') == v[:])
True
```

Change fill value for uninitialized chunks:

```
>>> a = zarr.full(10000, chunks=1000, fill_value=-1, dtype='i1')
>>> a[:]
array([-1, -1, -1, ..., -1, -1, -1], dtype=int8)
>>> v = a.view(fill_value=42)
>>> v[:]
array([42, 42, 42, ..., 42, 42, 42], dtype=int8)
```

Note that resizing or appending to views is not permitted:

```
>>> a = zarr.empty(10000)
>>> v = a.view()
>>> try:
...     v.resize(20000)
... except PermissionError as e:
...     print(e)
operation not permitted for views
```

3.3 Groups (zarr.hierarchy)

zarr.hierarchy.**group**(*store=None*, *overwrite=False*, *chunk_store=None*, *cache_attrs=True*, *synchronizer=None*, *path=None*, ***, *zarr_version=None*, *meta_array=None*)

Create a group.

Parameters

store
[MutableMapping or string, optional] Store or path to directory in file system.

overwrite
[bool, optional] If True, delete any pre-existing data in *store* at *path* before creating the group.

chunk_store
[MutableMapping, optional] Separate storage for chunks. If not provided, *store* will be used for storage of both chunks and metadata.

cache_attrs
[bool, optional] If True (default), user attributes will be cached for attribute read operations. If False, user attributes are reloaded from the store prior to all attribute read operations.

synchronizer
[object, optional] Array synchronizer.

path
[string, optional] Group path within store.

meta_array
[array-like, optional] An array instance to use for determining arrays to create and return to users. Use *numpy.empty(())* by default.

New in version 2.16.1.

Returns

g
[zarr.hierarchy.Group]

Examples

Create a group in memory:

```
>>> import zarr
>>> g = zarr.group()
>>> g
<zarr.hierarchy.Group '/'>
```

Create a group with a different store:

```
>>> store = zarr.DirectoryStore('data/example.zarr')
>>> g = zarr.group(store=store, overwrite=True)
>>> g
<zarr.hierarchy.Group '/'>
```

zarr.hierarchy.**open_group**(*store=None*, *mode='a'*, *cache_attrs=True*, *synchronizer=None*, *path=None*, *chunk_store=None*, *storage_options=None*, ***, *zarr_version=None*, *meta_array=None*)

Open a group using file-mode-like semantics.

Parameters

store
[MutableMapping or string, optional] Store or path to directory in file system or name of zip file.

mode
[{'r', 'r+', 'a', 'w', 'w-'}, optional] Persistence mode: 'r' means read only (must exist); 'r+' means read/write (must exist); 'a' means read/write (create if doesn't exist); 'w' means create (overwrite if exists); 'w-' means create (fail if exists).

cache_attrs
[bool, optional] If True (default), user attributes will be cached for attribute read operations. If False, user attributes are reloaded from the store prior to all attribute read operations.

synchronizer
[object, optional] Array synchronizer.

path
[string, optional] Group path within store.

chunk_store
[MutableMapping or string, optional] Store or path to directory in file system or name of zip file.

storage_options
[dict] If using an fsspec URL to create the store, these will be passed to the backend implementation. Ignored otherwise.

meta_array
[array-like, optional] An array instance to use for determining arrays to create and return to users. Use *numpy.empty(())* by default.

New in version 2.13.

Returns

g
[zarr.hierarchy.Group]

Examples

```
>>> import zarr
>>> root = zarr.open_group('data/example.zarr', mode='w')
>>> foo = root.create_group('foo')
>>> bar = root.create_group('bar')
>>> root
<zarr.hierarchy.Group '/'>
>>> root2 = zarr.open_group('data/example.zarr', mode='a')
>>> root2
<zarr.hierarchy.Group '/'>
>>> root == root2
True
```

class zarr.hierarchy.**Group**(*store*, *path=None*, *read_only=False*, *chunk_store=None*, *cache_attrs=True*, *synchronizer=None*, *zarr_version=None*, ***, *meta_array=None*)

Instantiate a group from an initialized store.

Parameters

store
[MutableMapping] Group store, already initialized. If the Group is used in a context manager, and the store has a `close` method, it will be called on exit.

path
[string, optional] Group path.

read_only
[bool, optional] True if group should be protected against modification.

chunk_store
[MutableMapping, optional] Separate storage for chunks. If not provided, *store* will be used for storage of both chunks and metadata.

cache_attrs
[bool, optional] If True (default), user attributes will be cached for attribute read operations. If False, user attributes are reloaded from the store prior to all attribute read operations.

synchronizer
[object, optional] Array synchronizer.

meta_array
[array-like, optional] An array instance to use for determining arrays to create and return to users. Use *numpy.empty(())* by default.

New in version 2.13.

Attributes

store
A MutableMapping providing the underlying storage for the group.

path
Storage path.

name
Group name following h5py convention.

read_only
A boolean, True if modification operations are not permitted.

chunk_store
A MutableMapping providing the underlying storage for array chunks.

synchronizer
Object used to synchronize write access to groups and arrays.

attrs
A MutableMapping containing user-defined attributes.

info
Return diagnostic information about the group.

meta_array
An array-like instance to use for determining arrays to create and return to users.

Methods

__len__()	Number of members.
__iter__()	Return an iterator over group member names.
__contains__(item)	Test for group membership.
__getitem__(item)	Obtain a group member.
__enter__()	Return the Group for use as a context manager.
__exit__(exc_type, exc_val, exc_tb)	Call the close method of the underlying Store.
group_keys()	Return an iterator over member names for groups only.
groups()	Return an iterator over (name, value) pairs for groups only.
array_keys([recurse])	Return an iterator over member names for arrays only.
arrays([recurse])	Return an iterator over (name, value) pairs for arrays only.
visit(func)	Run `func` on each object's path.
visitkeys(func)	An alias for *visit()*.
visitvalues(func)	Run `func` on each object.
visititems(func)	Run `func` on each object's path and the object itself.
tree([expand, level])	Provide a `print`-able display of the hierarchy.
create_group(name[, overwrite])	Create a sub-group.
require_group(name[, overwrite])	Obtain a sub-group, creating one if it doesn't exist.
create_groups(*names, **kwargs)	Convenience method to create multiple groups in a single call.
require_groups(*names)	Convenience method to require multiple groups in a single call.
create_dataset(name, **kwargs)	Create an array.
require_dataset(name, shape[, dtype, exact])	Obtain an array, creating if it doesn't exist.

continues on next page

Table 2 – continued from previous page

create(name, **kwargs)	Create an array.
empty(name, **kwargs)	Create an array.
zeros(name, **kwargs)	Create an array.
ones(name, **kwargs)	Create an array.
full(name, fill_value, **kwargs)	Create an array.
array(name, data, **kwargs)	Create an array.
empty_like(name, data, **kwargs)	Create an array.
zeros_like(name, data, **kwargs)	Create an array.
ones_like(name, data, **kwargs)	Create an array.
full_like(name, data, **kwargs)	Create an array.
info	Return diagnostic information about the group.
move(source, dest)	Move contents from one path to another relative to the Group.

__len__()

Number of members.

__iter__()

Return an iterator over group member names.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> d1 = g1.create_dataset('baz', shape=100, chunks=10)
>>> d2 = g1.create_dataset('quux', shape=200, chunks=20)
>>> for name in g1:
...     print(name)
bar
baz
foo
quux
```

__contains__(*item*)

Test for group membership.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> d1 = g1.create_dataset('bar', shape=100, chunks=10)
>>> 'foo' in g1
True
>>> 'bar' in g1
True
>>> 'baz' in g1
False
```

__getitem__(*item*)

Obtain a group member.

Parameters

item
[string] Member name or path.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> d1 = g1.create_dataset('foo/bar/baz', shape=100, chunks=10)
>>> g1['foo']
<zarr.hierarchy.Group '/foo'>
>>> g1['foo/bar']
<zarr.hierarchy.Group '/foo/bar'>
>>> g1['foo/bar/baz']
<zarr.core.Array '/foo/bar/baz' (100,) float64>
```

__enter__()

Return the Group for use as a context manager.

__exit__(*exc_type*, *exc_val*, *exc_tb*)

Call the close method of the underlying Store.

group_keys()

Return an iterator over member names for groups only.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> d1 = g1.create_dataset('baz', shape=100, chunks=10)
>>> d2 = g1.create_dataset('quux', shape=200, chunks=20)
>>> sorted(g1.group_keys())
['bar', 'foo']
```

groups()

Return an iterator over (name, value) pairs for groups only.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> d1 = g1.create_dataset('baz', shape=100, chunks=10)
>>> d2 = g1.create_dataset('quux', shape=200, chunks=20)
>>> for n, v in g1.groups():
...     print(n, type(v))
bar <class 'zarr.hierarchy.Group'>
foo <class 'zarr.hierarchy.Group'>
```

array_keys(*recurse=False*)

Return an iterator over member names for arrays only.

Parameters

recurse
[recurse, optional] Option to return member names for all arrays, even from groups below the current one. If False, only member names for arrays in the current group will be returned. Default value is False.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> d1 = g1.create_dataset('baz', shape=100, chunks=10)
>>> d2 = g1.create_dataset('quux', shape=200, chunks=20)
>>> sorted(g1.array_keys())
['baz', 'quux']
```

arrays(*recurse=False*)

Return an iterator over (name, value) pairs for arrays only.

Parameters

recurse
[recurse, optional] Option to return (name, value) pairs for all arrays, even from groups below the current one. If False, only (name, value) pairs for arrays in the current group will be returned. Default value is False.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> d1 = g1.create_dataset('baz', shape=100, chunks=10)
>>> d2 = g1.create_dataset('quux', shape=200, chunks=20)
>>> for n, v in g1.arrays():
...     print(n, type(v))
baz <class 'zarr.core.Array'>
quux <class 'zarr.core.Array'>
```

visit(*func*)

Run func on each object's path.

Note: If func returns None (or doesn't return),
iteration continues. However, if func returns anything else, it ceases and returns that value.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> g4 = g3.create_group('baz')
>>> g5 = g3.create_group('quux')
>>> def print_visitor(name):
...     print(name)
>>> g1.visit(print_visitor)
bar
bar/baz
bar/quux
foo
>>> g3.visit(print_visitor)
baz
quux
```

Search for members matching some name query can be implemented using visit that is, find and findall. Consider the following tree:

```
/
├── aaa
│   └── bbb
│       └── ccc
│           └── aaa
├── bar
└── foo
```

It is created as follows:

```
>>> root = zarr.group()
>>> foo = root.create_group("foo")
```

(continues on next page)

(continued from previous page)

```
>>> bar = root.create_group("bar")
>>> root.create_group("aaa").create_group("bbb").create_group("ccc").create_
↪group("aaa")
<zarr.hierarchy.Group '/aaa/bbb/ccc/aaa'>
```

For `find`, the first path that matches a given pattern (for example "aaa") is returned. Note that a non-None value is returned in the visit function to stop further iteration.

```
>>> import re
>>> pattern = re.compile("aaa")
>>> found = None
>>> def find(path):
...     global found
...     if pattern.search(path) is not None:
...         found = path
...         return True
...
>>> root.visit(find)
True
>>> print(found)
aaa
```

For `findall`, all the results are gathered into a list

```
>>> pattern = re.compile("aaa")
>>> found = []
>>> def findall(path):
...     if pattern.search(path) is not None:
...         found.append(path)
...
>>> root.visit(findall)
>>> print(found)
['aaa', 'aaa/bbb', 'aaa/bbb/ccc', 'aaa/bbb/ccc/aaa']
```

To match only on the last part of the path, use a greedy regex to filter out the prefix:

```
>>> prefix_pattern = re.compile(r".*/")
>>> pattern = re.compile("aaa")
>>> found = []
>>> def findall(path):
...     match = prefix_pattern.match(path)
...     if match is None:
...         name = path
...     else:
...         _, end = match.span()
...         name = path[end:]
...     if pattern.search(name) is not None:
...         found.append(path)
...     return None
...
>>> root.visit(findall)
>>> print(found)
['aaa', 'aaa/bbb/ccc/aaa']
```

visitkeys(*func*)

An alias for *visit()*.

visitvalues(*func*)

Run func on each object.

Note: If func returns None (or doesn't return),
iteration continues. However, if func returns anything else, it ceases and returns that value.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> g4 = g3.create_group('baz')
>>> g5 = g3.create_group('quux')
>>> def print_visitor(obj):
...     print(obj)
>>> g1.visitvalues(print_visitor)
<zarr.hierarchy.Group '/bar'>
<zarr.hierarchy.Group '/bar/baz'>
<zarr.hierarchy.Group '/bar/quux'>
<zarr.hierarchy.Group '/foo'>
>>> g3.visitvalues(print_visitor)
<zarr.hierarchy.Group '/bar/baz'>
<zarr.hierarchy.Group '/bar/quux'>
```

visititems(*func*)

Run func on each object's path and the object itself.

Note: If func returns None (or doesn't return),
iteration continues. However, if func returns anything else, it ceases and returns that value.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> g4 = g3.create_group('baz')
>>> g5 = g3.create_group('quux')
>>> def print_visitor(name, obj):
...     print((name, obj))
>>> g1.visititems(print_visitor)
('bar', <zarr.hierarchy.Group '/bar'>)
('bar/baz', <zarr.hierarchy.Group '/bar/baz'>)
('bar/quux', <zarr.hierarchy.Group '/bar/quux'>)
('foo', <zarr.hierarchy.Group '/foo'>)
>>> g3.visititems(print_visitor)
('baz', <zarr.hierarchy.Group '/bar/baz'>)
('quux', <zarr.hierarchy.Group '/bar/quux'>)
```

tree(*expand=False*, *level=None*)

Provide a `print`-able display of the hierarchy.

Parameters

expand
[bool, optional] Only relevant for HTML representation. If True, tree will be fully expanded.

level
[int, optional] Maximum depth to descend into hierarchy.

Notes

Please note that this is an experimental feature. The behaviour of this function is still evolving and the default output and/or parameters may change in future versions.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> g4 = g3.create_group('baz')
>>> g5 = g3.create_group('quux')
>>> d1 = g5.create_dataset('baz', shape=100, chunks=10)
>>> g1.tree()
/
 ├── bar
 │   ├── baz
 │   └── quux
 │       └── baz (100,) float64
 └── foo
>>> g1.tree(level=2)
/
 ├── bar
 │   ├── baz
 │   └── quux
 └── foo
>>> g3.tree()
bar
 ├── baz
 └── quux
     └── baz (100,) float64
```

create_group(*name*, *overwrite=False*)

Create a sub-group.

Parameters

name
[string] Group name.

overwrite
[bool, optional] If True, overwrite any existing array with the given name.

Returns

g
[zarr.hierarchy.Group]

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> g4 = g1.create_group('baz/quux')
```

require_group(*name*, *overwrite=False*)

Obtain a sub-group, creating one if it doesn't exist.

Parameters

name
[string] Group name.

overwrite
[bool, optional] Overwrite any existing array with given *name* if present.

Returns

g
[zarr.hierarchy.Group]

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.require_group('foo')
>>> g3 = g1.require_group('foo')
>>> g2 == g3
True
```

create_groups(**names*, ***kwargs*)

Convenience method to create multiple groups in a single call.

require_groups(**names*)

Convenience method to require multiple groups in a single call.

create_dataset(*name*, ***kwargs*)

Create an array.

Arrays are known as "datasets" in HDF5 terminology. For compatibility with h5py, Zarr groups also implement the require_dataset() method.

Parameters

name
[string] Array name.

data
[array-like, optional] Initial data.

shape
[int or tuple of ints] Array shape.

chunks
[int or tuple of ints, optional] Chunk shape. If not provided, will be guessed from *shape* and *dtype*.

dtype
[string or dtype, optional] NumPy dtype.

compressor
[Codec, optional] Primary compressor.

fill_value
[object] Default value to use for uninitialized portions of the array.

order
[{'C', 'F'}, optional] Memory layout to be used within each chunk.

synchronizer
[zarr.sync.ArraySynchronizer, optional] Array synchronizer.

filters
[sequence of Codecs, optional] Sequence of filters to use to encode chunk data prior to compression.

overwrite
[bool, optional] If True, replace any existing array or group with the given name.

cache_metadata
[bool, optional] If True, array configuration metadata will be cached for the lifetime of the object. If False, array metadata will be reloaded prior to all data access and modification operations (may incur overhead depending on storage and data access pattern).

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

Returns

a
[zarr.core.Array]

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> d1 = g1.create_dataset('foo', shape=(10000, 10000),
...                        chunks=(1000, 1000))
>>> d1
<zarr.core.Array '/foo' (10000, 10000) float64>
>>> d2 = g1.create_dataset('bar/baz/qux', shape=(100, 100, 100),
...                        chunks=(100, 10, 10))
>>> d2
<zarr.core.Array '/bar/baz/qux' (100, 100, 100) float64>
```

require_dataset(*name*, *shape*, *dtype=None*, *exact=False*, ***kwargs*)

Obtain an array, creating if it doesn't exist.

Arrays are known as "datasets" in HDF5 terminology. For compatibility with h5py, Zarr groups also implement the create_dataset() method.

Other *kwargs* are as per *zarr.hierarchy.Group.create_dataset()*.

Parameters

name
[string] Array name.

shape
[int or tuple of ints] Array shape.

dtype
[string or dtype, optional] NumPy dtype.

exact
[bool, optional] If True, require *dtype* to match exactly. If false, require *dtype* can be cast from array dtype.

create(*name*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.create()*.

empty(*name*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.empty()*.

zeros(*name*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.zeros()*.

ones(*name*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.ones()*.

full(*name*, *fill_value*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.full()*.

array(*name*, *data*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.array()*.

empty_like(*name*, *data*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.empty_like()*.

zeros_like(*name*, *data*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.zeros_like()*.

ones_like(*name*, *data*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.ones_like()*.

full_like(*name*, *data*, ***kwargs*)

Create an array. Keyword arguments as per *zarr.creation.full_like()*.

move(*source*, *dest*)

Move contents from one path to another relative to the Group.

Parameters

source
[string] Name or path to a Zarr object to move.

dest
[string] New name or path of the Zarr object.

3.4 Storage (zarr.storage)

This module contains storage classes for use with Zarr arrays and groups.

Note that any object implementing the MutableMapping interface from the collections module in the Python standard library can be used as a Zarr array store, as long as it accepts string (str) keys and bytes values.

In addition to the MutableMapping interface, store classes may also implement optional methods *listdir* (list members of a "directory") and *rmdir* (remove all members of a "directory"). These methods should be implemented if the store class is aware of the hierarchical organisation of resources within the store and can provide efficient implementations. If these methods are not available, Zarr will fall back to slower implementations that work via the MutableMapping interface. Store classes may also optionally implement a *rename* method (rename all members under a given path) and a *getsize* method (return the size in bytes of a given value).

class zarr.storage.**MemoryStore**(*root=None*, *cls=<class 'dict'>*, *dimension_separator=None*)

Store class that uses a hierarchy of KVStore objects, thus all data will be held in main memory.

Notes

Safe to write in multiple threads.

Examples

This is the default class used when creating a group. E.g.:

```
>>> import zarr
>>> g = zarr.group()
>>> type(g.store)
<class 'zarr.storage.MemoryStore'>
```

Note that the default class when creating an array is the built-in KVStore class, i.e.:

```
>>> z = zarr.zeros(100)
>>> type(z.store)
<class 'zarr.storage.KVStore'>
```

class zarr.storage.**DirectoryStore**(*path*, *normalize_keys=False*, *dimension_separator: Literal['.', '/'] | None = None*)

Storage class using directories and files on a standard file system.

Parameters

path
[string] Location of directory to use as the root of the storage hierarchy.

normalize_keys
[bool, optional] If True, all store keys will be normalized to use lower case characters (e.g. 'foo' and 'FOO' will be treated as equivalent). This can be useful to avoid potential discrepancies between case-sensitive and case-insensitive file system. Default value is False.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

Notes

Atomic writes are used, which means that data are first written to a temporary file, then moved into place when the write is successfully completed. Files are only held open while they are being read or written and are closed immediately afterwards, so there is no need to manually close any files.

Safe to write in multiple threads or processes.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.DirectoryStore('data/array.zarr')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
```

Each chunk of the array is stored as a separate file on the file system, i.e.:

```
>>> import os
>>> sorted(os.listdir('data/array.zarr'))
['.zarray', '0.0', '0.1', '1.0', '1.1']
```

Store a group:

```
>>> store = zarr.DirectoryStore('data/group.zarr')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
```

When storing a group, levels in the group hierarchy will correspond to directories on the file system, i.e.:

```
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup', 'foo']
>>> sorted(os.listdir('data/group.zarr/foo'))
['.zgroup', 'bar']
>>> sorted(os.listdir('data/group.zarr/foo/bar'))
['.zarray', '0.0', '0.1', '1.0', '1.1']
```

class zarr.storage.**TempStore**(*suffix=''*, *prefix='zarr'*, *dir=None*, *normalize_keys=False*, *dimension_separator: Literal['.', '/'] | None = None*)

Directory store using a temporary directory for storage.

Parameters

suffix
[string, optional] Suffix for the temporary directory name.

prefix
[string, optional] Prefix for the temporary directory name.

dir
[string, optional] Path to parent directory in which to create temporary directory.

normalize_keys
[bool, optional] If True, all store keys will be normalized to use lower case characters (e.g.

'foo' and 'FOO' will be treated as equivalent). This can be useful to avoid potential discrepancies between case-sensitive and case-insensitive file system. Default value is False.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

class zarr.storage.**NestedDirectoryStore**(*path*, *normalize_keys=False*, *dimension_separator: Literal['.', '/'] | None = '/'*)

Storage class using directories and files on a standard file system, with special handling for chunk keys so that chunk files for multidimensional arrays are stored in a nested directory tree.

Parameters

path
[string] Location of directory to use as the root of the storage hierarchy.

normalize_keys
[bool, optional] If True, all store keys will be normalized to use lower case characters (e.g. 'foo' and 'FOO' will be treated as equivalent). This can be useful to avoid potential discrepancies between case-sensitive and case-insensitive file system. Default value is False.

dimension_separator
[{'/'}, optional] Separator placed between the dimensions of a chunk. Only supports "/" unlike other implementations.

Notes

The *DirectoryStore* class stores all chunk files for an array together in a single directory. On some file systems, the potentially large number of files in a single directory can cause performance issues. The *NestedDirectoryStore* class provides an alternative where chunk files for multidimensional arrays will be organised into a directory hierarchy, thus reducing the number of files in any one directory.

Safe to write in multiple threads or processes.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.NestedDirectoryStore('data/array.zarr')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
```

Each chunk of the array is stored as a separate file on the file system, note the multiple directory levels used for the chunk files:

```
>>> import os
>>> sorted(os.listdir('data/array.zarr'))
['.zarray', '0', '1']
>>> sorted(os.listdir('data/array.zarr/0'))
['0', '1']
>>> sorted(os.listdir('data/array.zarr/1'))
['0', '1']
```

Store a group:

```
>>> store = zarr.NestedDirectoryStore('data/group.zarr')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
```

When storing a group, levels in the group hierarchy will correspond to directories on the file system, i.e.:

```
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup', 'foo']
>>> sorted(os.listdir('data/group.zarr/foo'))
['.zgroup', 'bar']
>>> sorted(os.listdir('data/group.zarr/foo/bar'))
['.zarray', '0', '1']
>>> sorted(os.listdir('data/group.zarr/foo/bar/0'))
['0', '1']
>>> sorted(os.listdir('data/group.zarr/foo/bar/1'))
['0', '1']
```

class zarr.storage.**ZipStore**(*path*, *compression=0*, *allowZip64=True*, *mode='a'*, *dimension_separator: Literal['.', '/'] | None = None*)

Storage class using a Zip file.

Parameters

path
[string] Location of file.

compression
[integer, optional] Compression method to use when writing to the archive.

allowZip64
[bool, optional] If True (the default) will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 2 GiB. If False will raise an exception when the ZIP file would require ZIP64 extensions.

mode
[string, optional] One of 'r' to read an existing file, 'w' to truncate and write a new file, 'a' to append to an existing file, or 'x' to exclusively create and write a new file.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

Notes

Each chunk of an array is stored as a separate entry in the Zip file. Note that Zip files do not provide any way to remove or replace existing entries. If an attempt is made to replace an entry, then a warning is generated by the Python standard library about a duplicate Zip file entry. This can be triggered if you attempt to write data to a Zarr array more than once, e.g.:

```
>>> store = zarr.ZipStore('data/example.zip', mode='w')
>>> z = zarr.zeros(100, chunks=10, store=store)
>>> # first write OK
... z[...] = 42
>>> # second write generates warnings
```

(continues on next page)

(continued from previous page)

```
... z[...] = 42
>>> store.close()
```

This can also happen in a more subtle situation, where data are written only once to a Zarr array, but the write operations are not aligned with chunk boundaries, e.g.:

```
>>> store = zarr.ZipStore('data/example.zip', mode='w')
>>> z = zarr.zeros(100, chunks=10, store=store)
>>> z[5:15] = 42
>>> # write overlaps chunk previously written, generates warnings
... z[15:25] = 42
```

To avoid creating duplicate entries, only write data once, and align writes with chunk boundaries. This alignment is done automatically if you call `z[...] = ...` or create an array from existing data via `zarr.array()`.

Alternatively, use a *DirectoryStore* when writing the data, then manually Zip the directory and use the Zip file for subsequent reads. Take note that the files in the Zip file must be relative to the root of the Zarr archive. You may find it easier to create such a Zip file with `7z`, e.g.:

```
7z a -tzip archive.zarr.zip archive.zarr/.
```

Safe to write in multiple threads but not in multiple processes.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.ZipStore('data/array.zip', mode='w')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.ZipStore('data/group.zip', mode='w')
>>> root = zarr.group(store=store)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

After modifying a ZipStore, the `close()` method must be called, otherwise essential data will not be written to the underlying Zip file. The ZipStore class also supports the context manager protocol, which ensures the `close()` method is called on leaving the context, e.g.:

```
>>> with zarr.ZipStore('data/array.zip', mode='w') as store:
...     z = zarr.zeros((10, 10), chunks=(5, 5), store=store)
...     z[...] = 42
...     # no need to call store.close()
```

close()

Closes the underlying zip file, ensuring all records are written.

flush()

Closes the underlying zip file, ensuring all records are written, then re-opens the file for further modifications.

class zarr.storage.**DBMStore**(*path*, *flag='c'*, *mode=438*, *open=None*, *write_lock=True*, *dimension_separator: Literal['.', '/'] | None = None*, ***open_kwargs*)

Storage class using a DBM-style database.

Parameters

path
[string] Location of database file.

flag
[string, optional] Flags for opening the database file.

mode
[int] File mode used if a new file is created.

open
[function, optional] Function to open the database file. If not provided, `dbm.open()` will be used on Python 3, and `anydbm.open()` will be used on Python 2.

write_lock: bool, optional
Use a lock to prevent concurrent writes from multiple threads (True by default).

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.e

****open_kwargs**
Keyword arguments to pass the *open* function.

Notes

Please note that, by default, this class will use the Python standard library *dbm.open* function to open the database file (or *anydbm.open* on Python 2). There are up to three different implementations of DBM-style databases available in any Python installation, and which one is used may vary from one system to another. Database file formats are not compatible between these different implementations. Also, some implementations are more efficient than others. In particular, the "dumb" implementation will be the fall-back on many systems, and has very poor performance for some usage scenarios. If you want to ensure a specific implementation is used, pass the corresponding open function, e.g., *dbm.gnu.open* to use the GNU DBM library.

Safe to write in multiple threads. May be safe to write in multiple processes, depending on which DBM implementation is being used, although this has not been tested.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.DBMStore('data/array.db')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.DBMStore('data/group.db')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

After modifying a DBMStore, the `close()` method must be called, otherwise essential data may not be written to the underlying database file. The DBMStore class also supports the context manager protocol, which ensures the `close()` method is called on leaving the context, e.g.:

```
>>> with zarr.DBMStore('data/array.db') as store:
...     z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
...     z[...] = 42
...     # no need to call store.close()
```

A different database library can be used by passing a different function to the *open* parameter. For example, if the bsddb3 package is installed, a Berkeley DB database can be used:

```
>>> import bsddb3
>>> store = zarr.DBMStore('data/array.bdb', open=bsddb3.btopen)
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()
```

close()

Closes the underlying database file.

flush()

Synchronizes data to the underlying database file.

class zarr.storage.**LMDBStore**(*path*, *buffers=True*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using LMDB. Requires the lmdb package to be installed.

Parameters

path
[string] Location of database file.

buffers
[bool, optional] If True (default) use support for buffers, which should increase performance by reducing memory copies.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *lmdb.open* function.

Notes

By default writes are not immediately flushed to disk to increase performance. You can ensure data are flushed to disk by calling the `flush()` or `close()` methods.

Should be safe to write in multiple threads or processes due to the synchronization support within LMDB, although writing from multiple processes has not been tested.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.LMDBStore('data/array.mdb')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.LMDBStore('data/group.mdb')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

After modifying a DBMStore, the `close()` method must be called, otherwise essential data may not be written to the underlying database file. The DBMStore class also supports the context manager protocol, which ensures the `close()` method is called on leaving the context, e.g.:

```
>>> with zarr.LMDBStore('data/array.mdb') as store:
...     z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
...     z[...] = 42
...     # no need to call store.close()
```

close()

Closes the underlying database.

flush()

Synchronizes data to the file system.

class zarr.storage.**SQLiteStore**(*path*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using SQLite.

Parameters

path
[string] Location of database file.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *sqlite3.connect* function.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.SQLiteStore('data/array.sqldb')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.SQLiteStore('data/group.sqldb')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

close()

Closes the underlying database.

class zarr.storage.**MongoDBStore**(*database='mongodb_zarr'*, *collection='zarr_collection'*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using MongoDB.

Note: This is an experimental feature.

Requires the pymongo package to be installed.

Parameters

database
[string] Name of database

collection
[string] Name of collection

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *pymongo.MongoClient* function.

Notes

The maximum chunksize in MongoDB documents is 16 MB.

class zarr.storage.**RedisStore**(*prefix='zarr'*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using Redis.

Note: This is an experimental feature.

Requires the redis package to be installed.

Parameters

prefix
[string] Name of prefix for Redis keys

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *redis.Redis* function.

class zarr.storage.**LRUStoreCache**(*store: BaseStore | MutableMapping*, *max_size: int*)

Storage class that implements a least-recently-used (LRU) cache layer over some other store. Intended primarily for use with stores that can be slow to access, e.g., remote stores that require network communication to store and retrieve data.

Parameters

store
[Store] The store containing the actual data to be cached.

max_size
[int] The maximum size that the cache may grow to, in number of bytes. Provide *None* if you would like the cache to have unlimited size.

Examples

The example below wraps an S3 store with an LRU cache:

```
>>> import s3fs
>>> import zarr
>>> s3 = s3fs.S3FileSystem(anon=True, client_kwargs=dict(region_name='eu-west-2'))
>>> store = s3fs.S3Map(root='zarr-demo/store', s3=s3, check=False)
>>> cache = zarr.LRUStoreCache(store, max_size=2**28)
>>> root = zarr.group(store=cache)
>>> z = root['foo/bar/baz']
>>> from timeit import timeit
>>> # first data access is relatively slow, retrieved from store
... timeit('print(z[:].tobytes())', number=1, globals=globals())
b'Hello from the cloud!'
0.1081731989979744
>>> # second data access is faster, uses cache
... timeit('print(z[:].tobytes())', number=1, globals=globals())
b'Hello from the cloud!'
0.0009490990014455747
```

invalidate()

Completely clear the cache.

invalidate_values()

Clear the values cache.

invalidate_keys()

Clear the keys cache.

class zarr.storage.**ABSStore**(*container=None*, *prefix=''*, *account_name=None*, *account_key=None*, *blob_service_kwargs=None*, *dimension_separator: Literal['.', '/'] | None = None*, *client=None*)

Storage class using Azure Blob Storage (ABS).

Parameters

container
[string] The name of the ABS container to use.

Deprecated since version Use: `client` instead.

prefix
[string] Location of the "directory" to use as the root of the storage hierarchy within the container.

account_name
[string] The Azure blob storage account name.

Deprecated since version 2.8.3: Use `client` instead.

account_key
[string] The Azure blob storage account access key.

Deprecated since version 2.8.3: Use `client` instead.

blob_service_kwargs
[dictionary] Extra arguments to be passed into the azure blob client, for e.g. when using the emulator, pass in blob_service_kwargs={'is_emulated': True}.

Deprecated since version 2.8.3: Use `client` instead.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

client
[azure.storage.blob.ContainerClient, optional] And `azure.storage.blob.ContainerClient` to connect with. See here # noqa for more.

New in version 2.8.3.

Notes

In order to use this store, you must install the Microsoft Azure Storage SDK for Python, `azure-storage-blob>=12.5.0`.

`class zarr.storage.FSStore`(*url*, *normalize_keys=False*, *key_separator=None*, *mode='w'*, *exceptions=(<class 'KeyError'>, <class 'PermissionError'>, <class 'OSError'>)*, *dimension_separator: ~typing.Literal['.', '/'] | None = None*, *fs=None*, *check=False*, *create=False*, *missing_exceptions=None*, ***storage_options*)

Wraps an fsspec.FSMap to give access to arbitrary filesystems

Requires that `fsspec` is installed, as well as any additional requirements for the protocol chosen.

Parameters

url
[str] The destination to map. If no fs is provided, should include protocol and path, like "s3://bucket/root". If an fs is provided, can be a path within that filesystem, like "bucket/root"

normalize_keys
[bool]

key_separator
[str] public API for accessing dimension_separator. Never *None* See dimension_separator for more information.

mode
[str] "w" for writable, "r" for read-only

exceptions
[list of Exception subclasses] When accessing data, any of these exceptions will be treated as a missing key

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

fs
[fsspec.spec.AbstractFileSystem, optional] An existing filesystem to use for the store.

check
[bool, optional] If True, performs a touch at the root location, to check for write access. Passed to *fsspec.mapping.FSMap* constructor.

create
[bool, optional] If True, performs a mkdir at the rool location. Passed to *fsspec.mapping.FSMap* constructor.

missing_exceptions
[sequence of Exceptions, optional] Exceptions classes to associate with missing files. Passed to *fsspec.mapping.FSMap* constructor.

storage_options
[passed to the fsspec implementation. Cannot be used] together with fs.

`class zarr.storage.ConsolidatedMetadataStore(`*store: BaseStore | MutableMapping*, *metadata_key='.zmetadata'*`)`

A layer over other storage, where the metadata has been consolidated into a single key.

The purpose of this class, is to be able to get all of the metadata for a given array in a single read operation from the underlying storage. See *zarr.convenience.consolidate_metadata()* for how to create this single metadata key.

This class loads from the one key, and stores the data in a dict, so that accessing the keys no longer requires operations on the backend store.

This class is read-only, and attempts to change the array metadata will fail, but changing the data is possible. If the backend storage is changed directly, then the metadata stored here could become obsolete, and *zarr.convenience.consolidate_metadata()* should be called again and the class re-invoked. The use case is for write once, read many times.

New in version 2.3.

Note: This is an experimental feature.

Parameters

store: Store
Containing the zarr array.

metadata_key: str
The target in the store where all of the metadata are stored. We assume JSON encoding.

See also:

zarr.convenience.consolidate_metadata, *zarr.convenience.open_consolidated*

zarr.storage.**init_array**(*store: BaseStore | MutableMapping*, *shape: int | Tuple[int, ...]*, *chunks: bool | int | Tuple[int, ...] = True*, *dtype=None*, *compressor='default'*, *fill_value=None*, *order: str = 'C'*, *overwrite: bool = False*, *path: str | bytes | None = None*, *chunk_store: BaseStore | MutableMapping | None = None*, *filters=None*, *object_codec=None*, *dimension_separator: Literal['.', '/'] | None = None*, *storage_transformers=()*)

Initialize an array store with the given configuration. Note that this is a low-level function and there should be no need to call this directly from user code.

Parameters

store
[Store] A mapping that supports string keys and bytes-like values.

shape
[int or tuple of ints] Array shape.

chunks
[bool, int or tuple of ints, optional] Chunk shape. If True, will be guessed from *shape* and *dtype*. If False, will be set to *shape*, i.e., single chunk for the whole array.

dtype
[string or dtype, optional] NumPy dtype.

compressor
[Codec, optional] Primary compressor.

fill_value
[object] Default value to use for uninitialized portions of the array.

order
[{'C', 'F'}, optional] Memory layout to be used within each chunk.

overwrite
[bool, optional] If True, erase all data in *store* prior to initialisation.

path
[string, bytes, optional] Path under which array is stored.

chunk_store
[Store, optional] Separate storage for chunks. If not provided, *store* will be used for storage of both chunks and metadata.

filters
[sequence, optional] Sequence of filters to use to encode chunk data prior to compression.

object_codec
[Codec, optional] A codec to encode object arrays, only needed if dtype=object.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

Notes

The initialisation process involves normalising all array metadata, encoding as JSON and storing under the '.zarray' key.

Examples

Initialize an array store:

```
>>> from zarr.storage import init_array, KVStore
>>> store = KVStore(dict())
>>> init_array(store, shape=(10000, 10000), chunks=(1000, 1000))
>>> sorted(store.keys())
['.zarray']
```

Array metadata is stored as JSON:

```
>>> print(store['.zarray'].decode())
{
    "chunks": [
        1000,
        1000
    ],
    "compressor": {
        "blocksize": 0,
        "clevel": 5,
        "cname": "lz4",
        "id": "blosc",
        "shuffle": 1
    },
    "dtype": "<f8",
    "fill_value": null,
    "filters": null,
    "order": "C",
    "shape": [
        10000,
        10000
    ],
    "zarr_format": 2
}
```

Initialize an array using a storage path:

```
>>> store = KVStore(dict())
>>> init_array(store, shape=100000000, chunks=1000000, dtype='i1', path='foo')
>>> sorted(store.keys())
['.zgroup', 'foo/.zarray']
>>> print(store['foo/.zarray'].decode())
{
    "chunks": [
        1000000
    ],
    "compressor": {
```

(continues on next page)

(continued from previous page)

```
        "blocksize": 0,
        "clevel": 5,
        "cname": "lz4",
        "id": "blosc",
        "shuffle": 1
    },
    "dtype": "|i1",
    "fill_value": null,
    "filters": null,
    "order": "C",
    "shape": [
        100000000
    ],
    "zarr_format": 2
}
```

zarr.storage.**init_group**(*store: BaseStore | MutableMapping*, *overwrite: bool = False*, *path: str | bytes | None = None*, *chunk_store: BaseStore | MutableMapping | None = None*)

Initialize a group store. Note that this is a low-level function and there should be no need to call this directly from user code.

Parameters

store
[Store] A mapping that supports string keys and byte sequence values.

overwrite
[bool, optional] If True, erase all data in *store* prior to initialisation.

path
[string, optional] Path under which array is stored.

chunk_store
[Store, optional] Separate storage for chunks. If not provided, *store* will be used for storage of both chunks and metadata.

zarr.storage.**contains_array**(*store: BaseStore | MutableMapping*, *path: str | bytes | None = None*) → bool

Return True if the store contains an array at the given logical path.

zarr.storage.**contains_group**(*store: BaseStore | MutableMapping*, *path: str | bytes | None = None*, *explicit_only=True*) → bool

Return True if the store contains a group at the given logical path.

zarr.storage.**listdir**(*store: BaseStore*, *path: str | bytes | None = None*)

Obtain a directory listing for the given path. If *store* provides a *listdir* method, this will be called, otherwise will fall back to implementation via the *MutableMapping* interface.

zarr.storage.**rmdir**(*store: BaseStore | MutableMapping*, *path: str | bytes | None = None*)

Remove all items under the given path. If *store* provides a *rmdir* method, this will be called, otherwise will fall back to implementation via the *Store* interface.

zarr.storage.**getsize**(*store: BaseStore*, *path: str | bytes | None = None*) → int

Compute size of stored items for a given path. If *store* provides a *getsize* method, this will be called, otherwise will return -1.

zarr.storage.**rename**(*store: Store*, *src_path: str | bytes | None*, *dst_path: str | bytes | None*)

Rename all items under the given path. If *store* provides a *rename* method, this will be called, otherwise will fall back to implementation via the *Store* interface.

zarr.storage.**migrate_1to2**(*store*)

Migrate array metadata in *store* from Zarr format version 1 to version 2.

Parameters

store
[Store] Store to be migrated.

Notes

Version 1 did not support hierarchies, so this migration function will look for a single array in *store* and migrate the array metadata to version 2.

3.5 N5 (zarr.n5)

This module contains a storage class and codec to support the N5 format.

class zarr.n5.**N5Store**(*path*, *normalize_keys=False*, *dimension_separator: Literal['.', '/'] | None = '/'*)

Storage class using directories and files on a standard file system, following the N5 format (https://github.com/saalfeldlab/n5).

Parameters

path
[string] Location of directory to use as the root of the storage hierarchy.

normalize_keys
[bool, optional] If True, all store keys will be normalized to use lower case characters (e.g. 'foo' and 'FOO' will be treated as equivalent). This can be useful to avoid potential discrepancies between case-sensitive and case-insensitive file system. Default value is False.

Notes

This is an experimental feature.

Safe to write in multiple threads or processes.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.N5Store('data/array.n5')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
```

Store a group:

```
>>> store = zarr.N5Store('data/group.n5')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
```

3.6 Convenience functions (zarr.convenience)

Convenience functions for storing and loading data.

zarr.convenience.**open**(*store: BaseStore | MutableMapping | str | None = None*, *mode: str = 'a'*, ***, *zarr_version=None*, *path=None*, ***kwargs*)

Convenience function to open a group or array using file-mode-like semantics.

Parameters

store
[Store or string, optional] Store or path to directory in file system or name of zip file.

mode
[{'r', 'r+', 'a', 'w', 'w-'}, optional] Persistence mode: 'r' means read only (must exist); 'r+' means read/write (must exist); 'a' means read/write (create if doesn't exist); 'w' means create (overwrite if exists); 'w-' means create (fail if exists).

zarr_version
[{2, 3, None}, optional] The zarr protocol version to use. The default value of None will attempt to infer the version from *store* if possible, otherwise it will fall back to 2.

path
[str or None, optional] The path within the store to open.

****kwargs**
Additional parameters are passed through to *zarr.creation.open_array()* or *zarr.hierarchy.open_group()*.

Returns

z
[*zarr.core.Array* or *zarr.hierarchy.Group*] Array or group, depending on what exists in the given store.

See also:

zarr.creation.open_array, *zarr.hierarchy.open_group*

Examples

Storing data in a directory 'data/example.zarr' on the local file system:

```
>>> import zarr
>>> store = 'data/example.zarr'
>>> zw = zarr.open(store, mode='w', shape=100, dtype='i4')  # open new array
>>> zw
<zarr.core.Array (100,) int32>
>>> za = zarr.open(store, mode='a')  # open existing array for reading and writing
```

(continues on next page)

(continued from previous page)

```
>>> za
<zarr.core.Array (100,) int32>
>>> zr = zarr.open(store, mode='r')  # open existing array read-only
>>> zr
<zarr.core.Array (100,) int32 read-only>
>>> gw = zarr.open(store, mode='w')  # open new group, overwriting previous data
>>> gw
<zarr.hierarchy.Group '/'>
>>> ga = zarr.open(store, mode='a')  # open existing group for reading and writing
>>> ga
<zarr.hierarchy.Group '/'>
>>> gr = zarr.open(store, mode='r')  # open existing group read-only
>>> gr
<zarr.hierarchy.Group '/' read-only>
```

zarr.convenience.**save**(*store: BaseStore | MutableMapping | str | None*, **args*, *zarr_version=None*, *path=None*, ***kwargs*)

Convenience function to save an array or group of arrays to the local file system.

Parameters

store
[MutableMapping or string] Store or path to directory in file system or name of zip file.

args
[ndarray] NumPy arrays with data to save.

zarr_version
[{2, 3, None}, optional] The zarr protocol version to use when saving. The default value of None will attempt to infer the version from *store* if possible, otherwise it will fall back to 2.

path
[str or None, optional] The path within the group where the arrays will be saved.

kwargs
NumPy arrays with data to save.

See also:

save_array, *save_group*

Examples

Save an array to a directory on the file system (uses a `DirectoryStore`):

```
>>> import zarr
>>> import numpy as np
>>> arr = np.arange(10000)
>>> zarr.save('data/example.zarr', arr)
>>> zarr.load('data/example.zarr')
array([   0,    1,    2, ..., 9997, 9998, 9999])
```

Save an array to a Zip file (uses a `ZipStore`):

```
>>> zarr.save('data/example.zip', arr)
>>> zarr.load('data/example.zip')
array([   0,    1,    2, ..., 9997, 9998, 9999])
```

Save several arrays to a directory on the file system (uses a `DirectoryStore` and stores arrays in a group):

```
>>> import zarr
>>> import numpy as np
>>> a1 = np.arange(10000)
>>> a2 = np.arange(10000, 0, -1)
>>> zarr.save('data/example.zarr', a1, a2)
>>> loader = zarr.load('data/example.zarr')
>>> loader
<LazyLoader: arr_0, arr_1>
>>> loader['arr_0']
array([   0,    1,    2, ..., 9997, 9998, 9999])
>>> loader['arr_1']
array([10000,  9999,  9998, ...,     3,     2,     1])
```

Save several arrays using named keyword arguments:

```
>>> zarr.save('data/example.zarr', foo=a1, bar=a2)
>>> loader = zarr.load('data/example.zarr')
>>> loader
<LazyLoader: bar, foo>
>>> loader['foo']
array([   0,    1,    2, ..., 9997, 9998, 9999])
>>> loader['bar']
array([10000,  9999,  9998, ...,     3,     2,     1])
```

Store several arrays in a single zip file (uses a `ZipStore`):

```
>>> zarr.save('data/example.zip', foo=a1, bar=a2)
>>> loader = zarr.load('data/example.zip')
>>> loader
<LazyLoader: bar, foo>
>>> loader['foo']
array([   0,    1,    2, ..., 9997, 9998, 9999])
>>> loader['bar']
array([10000,  9999,  9998, ...,     3,     2,     1])
```

`zarr.convenience.load`(*store: BaseStore | MutableMapping | str | None*, *zarr_version=None*, *path=None*)

Load data from an array or group into memory.

Parameters

store
[MutableMapping or string] Store or path to directory in file system or name of zip file.

zarr_version
[{2, 3, None}, optional] The zarr protocol version to use when loading. The default value of None will attempt to infer the version from *store* if possible, otherwise it will fall back to 2.

path
[str or None, optional] The path within the store from which to load.

Returns

out
If the store contains an array, out will be a numpy array. If the store contains a group, out will be a dict-like object where keys are array names and values are numpy arrays.

See also:

save, savez

Notes

If loading data from a group of arrays, data will not be immediately loaded into memory. Rather, arrays will be loaded into memory as they are requested.

zarr.convenience.**save_array**(*store: BaseStore | MutableMapping | str | None*, *arr*, ***, *zarr_version=None*, *path=None*, ***kwargs*)

Convenience function to save a NumPy array to the local file system, following a similar API to the NumPy save() function.

Parameters

store
[MutableMapping or string] Store or path to directory in file system or name of zip file.

arr
[ndarray] NumPy array with data to save.

zarr_version
[{2, 3, None}, optional] The zarr protocol version to use when saving. The default value of None will attempt to infer the version from *store* if possible, otherwise it will fall back to 2.

path
[str or None, optional] The path within the store where the array will be saved.

kwargs
Passed through to `create()`, e.g., compressor.

Examples

Save an array to a directory on the file system (uses a `DirectoryStore`):

```
>>> import zarr
>>> import numpy as np
>>> arr = np.arange(10000)
>>> zarr.save_array('data/example.zarr', arr)
>>> zarr.load('data/example.zarr')
array([   0,    1,    2, ..., 9997, 9998, 9999])
```

Save an array to a single file (uses a `ZipStore`):

```
>>> zarr.save_array('data/example.zip', arr)
>>> zarr.load('data/example.zip')
array([   0,    1,    2, ..., 9997, 9998, 9999])
```

zarr.convenience.**save_group**(*store: BaseStore | MutableMapping | str | None*, **args*, *zarr_version=None*, *path=None*, ***kwargs*)

Convenience function to save several NumPy arrays to the local file system, following a similar API to the NumPy savez()/savez_compressed() functions.

Parameters

store
[MutableMapping or string] Store or path to directory in file system or name of zip file.

args
[ndarray] NumPy arrays with data to save.

zarr_version
[{2, 3, None}, optional] The zarr protocol version to use when saving. The default value of None will attempt to infer the version from *store* if possible, otherwise it will fall back to 2.

path
[str or None, optional] Path within the store where the group will be saved.

kwargs
NumPy arrays with data to save.

Notes

Default compression options will be used.

Examples

Save several arrays to a directory on the file system (uses a `DirectoryStore`):

```
>>> import zarr
>>> import numpy as np
>>> a1 = np.arange(10000)
>>> a2 = np.arange(10000, 0, -1)
>>> zarr.save_group('data/example.zarr', a1, a2)
>>> loader = zarr.load('data/example.zarr')
>>> loader
<LazyLoader: arr_0, arr_1>
>>> loader['arr_0']
array([   0,    1,    2, ..., 9997, 9998, 9999])
>>> loader['arr_1']
array([10000,  9999,  9998, ...,     3,     2,     1])
```

Save several arrays using named keyword arguments:

```
>>> zarr.save_group('data/example.zarr', foo=a1, bar=a2)
>>> loader = zarr.load('data/example.zarr')
>>> loader
<LazyLoader: bar, foo>
>>> loader['foo']
array([   0,    1,    2, ..., 9997, 9998, 9999])
>>> loader['bar']
array([10000,  9999,  9998, ...,     3,     2,     1])
```

Store several arrays in a single zip file (uses a `ZipStore`):

```
>>> zarr.save_group('data/example.zip', foo=a1, bar=a2)
>>> loader = zarr.load('data/example.zip')
>>> loader
<LazyLoader: bar, foo>
>>> loader['foo']
array([   0,    1,    2, ..., 9997, 9998, 9999])
>>> loader['bar']
array([10000,  9999,  9998, ...,     3,     2,     1])
```

zarr.convenience.**copy**(*source*, *dest*, *name=None*, *shallow=False*, *without_attrs=False*, *log=None*, *if_exists='raise'*, *dry_run=False*, ***create_kws*)

Copy the *source* array or group into the *dest* group.

Parameters

source
[group or array/dataset] A zarr group or array, or an h5py group or dataset.

dest
[group] A zarr or h5py group.

name
[str, optional] Name to copy the object to.

shallow
[bool, optional] If True, only copy immediate children of *source*.

without_attrs
[bool, optional] Do not copy user attributes.

log
[callable, file path or file-like object, optional] If provided, will be used to log progress information.

if_exists
[{'raise', 'replace', 'skip', 'skip_initialized'}, optional] How to handle arrays that already exist in the destination group. If 'raise' then a CopyError is raised on the first array already present in the destination group. If 'replace' then any array will be replaced in the destination. If 'skip' then any existing arrays will not be copied. If 'skip_initialized' then any existing arrays with all chunks initialized will not be copied (not available when copying to h5py).

dry_run
[bool, optional] If True, don't actually copy anything, just log what would have happened.

****create_kws**
Passed through to the create_dataset method when copying an array/dataset.

Returns

n_copied
[int] Number of items copied.

n_skipped
[int] Number of items skipped.

n_bytes_copied
[int] Number of bytes of data that were actually copied.

Notes

Please note that this is an experimental feature. The behaviour of this function is still evolving and the default behaviour and/or parameters may change in future versions.

Examples

Here's an example of copying a group named 'foo' from an HDF5 file to a Zarr group:

```
>>> import h5py
>>> import zarr
>>> import numpy as np
>>> source = h5py.File('data/example.h5', mode='w')
>>> foo = source.create_group('foo')
>>> baz = foo.create_dataset('bar/baz', data=np.arange(100), chunks=(50,))
>>> spam = source.create_dataset('spam', data=np.arange(100, 200), chunks=(30,))
>>> zarr.tree(source)
/
 ├── foo
 │   └── bar
 │       └── baz (100,) int64
 └── spam (100,) int64
>>> dest = zarr.group()
>>> from sys import stdout
>>> zarr.copy(source['foo'], dest, log=stdout)
copy /foo
copy /foo/bar
copy /foo/bar/baz (100,) int64
all done: 3 copied, 0 skipped, 800 bytes copied
(3, 0, 800)
>>> dest.tree()  # N.B., no spam
/
 └── foo
     └── bar
         └── baz (100,) int64
>>> source.close()
```

The `if_exists` parameter provides options for how to handle pre-existing data in the destination. Here are some examples of these options, also using `dry_run=True` to find out what would happen without actually copying anything:

```
>>> source = zarr.group()
>>> dest = zarr.group()
>>> baz = source.create_dataset('foo/bar/baz', data=np.arange(100))
>>> spam = source.create_dataset('foo/spam', data=np.arange(1000))
>>> existing_spam = dest.create_dataset('foo/spam', data=np.arange(1000))
>>> from sys import stdout
>>> try:
...     zarr.copy(source['foo'], dest, log=stdout, dry_run=True)
... except zarr.CopyError as e:
...     print(e)
...
copy /foo
```

(continues on next page)

(continued from previous page)

```
copy /foo/bar
copy /foo/bar/baz (100,) int64
an object 'spam' already exists in destination '/foo'
>>> zarr.copy(source['foo'], dest, log=stdout, if_exists='replace', dry_run=True)
copy /foo
copy /foo/bar
copy /foo/bar/baz (100,) int64
copy /foo/spam (1000,) int64
dry run: 4 copied, 0 skipped
(4, 0, 0)
>>> zarr.copy(source['foo'], dest, log=stdout, if_exists='skip', dry_run=True)
copy /foo
copy /foo/bar
copy /foo/bar/baz (100,) int64
skip /foo/spam (1000,) int64
dry run: 3 copied, 1 skipped
(3, 1, 0)
```

zarr.convenience.**copy_all**(*source*, *dest*, *shallow=False*, *without_attrs=False*, *log=None*, *if_exists='raise'*, *dry_run=False*, ***create_kws*)

Copy all children of the *source* group into the *dest* group.

Parameters

source
[group or array/dataset] A zarr group or array, or an h5py group or dataset.

dest
[group] A zarr or h5py group.

shallow
[bool, optional] If True, only copy immediate children of *source*.

without_attrs
[bool, optional] Do not copy user attributes.

log
[callable, file path or file-like object, optional] If provided, will be used to log progress information.

if_exists
[{'raise', 'replace', 'skip', 'skip_initialized'}, optional] How to handle arrays that already exist in the destination group. If 'raise' then a CopyError is raised on the first array already present in the destination group. If 'replace' then any array will be replaced in the destination. If 'skip' then any existing arrays will not be copied. If 'skip_initialized' then any existing arrays with all chunks initialized will not be copied (not available when copying to h5py).

dry_run
[bool, optional] If True, don't actually copy anything, just log what would have happened.

****create_kws**
Passed through to the create_dataset method when copying an array/dataset.

Returns

n_copied
[int] Number of items copied.

n_skipped
[int] Number of items skipped.

n_bytes_copied
[int] Number of bytes of data that were actually copied.

Notes

Please note that this is an experimental feature. The behaviour of this function is still evolving and the default behaviour and/or parameters may change in future versions.

Examples

```
>>> import h5py
>>> import zarr
>>> import numpy as np
>>> source = h5py.File('data/example.h5', mode='w')
>>> foo = source.create_group('foo')
>>> baz = foo.create_dataset('bar/baz', data=np.arange(100), chunks=(50,))
>>> spam = source.create_dataset('spam', data=np.arange(100, 200), chunks=(30,))
>>> zarr.tree(source)
/
 ├── foo
 │   └── bar
 │       └── baz (100,) int64
 └── spam (100,) int64
>>> dest = zarr.group()
>>> import sys
>>> zarr.copy_all(source, dest, log=sys.stdout)
copy /foo
copy /foo/bar
copy /foo/bar/baz (100,) int64
copy /spam (100,) int64
all done: 4 copied, 0 skipped, 1,600 bytes copied
(4, 0, 1600)
>>> dest.tree()
/
 ├── foo
 │   └── bar
 │       └── baz (100,) int64
 └── spam (100,) int64
>>> source.close()
```

zarr.convenience.**copy_store**(*source*, *dest*, *source_path=''*, *dest_path=''*, *excludes=None*, *includes=None*, *flags=0*, *if_exists='raise'*, *dry_run=False*, *log=None*)

Copy data directly from the *source* store to the *dest* store. Use this function when you want to copy a group or array in the most efficient way, preserving all configuration and attributes. This function is more efficient than the copy() or copy_all() functions because it avoids de-compressing and re-compressing data, rather the compressed chunk data for each array are copied directly between stores.

Parameters

source
[Mapping] Store to copy data from.

dest
[MutableMapping] Store to copy data into.

source_path
[str, optional] Only copy data from under this path in the source store.

dest_path
[str, optional] Copy data into this path in the destination store.

excludes
[sequence of str, optional] One or more regular expressions which will be matched against keys in the source store. Any matching key will not be copied.

includes
[sequence of str, optional] One or more regular expressions which will be matched against keys in the source store and will override any excludes also matching.

flags
[int, optional] Regular expression flags used for matching excludes and includes.

if_exists
[{'raise', 'replace', 'skip'}, optional] How to handle keys that already exist in the destination store. If 'raise' then a CopyError is raised on the first key already present in the destination store. If 'replace' then any data will be replaced in the destination. If 'skip' then any existing keys will not be copied.

dry_run
[bool, optional] If True, don't actually copy anything, just log what would have happened.

log
[callable, file path or file-like object, optional] If provided, will be used to log progress information.

Returns

n_copied
[int] Number of items copied.

n_skipped
[int] Number of items skipped.

n_bytes_copied
[int] Number of bytes of data that were actually copied.

Notes

Please note that this is an experimental feature. The behaviour of this function is still evolving and the default behaviour and/or parameters may change in future versions.

Examples

```
>>> import zarr
>>> store1 = zarr.DirectoryStore('data/example.zarr')
>>> root = zarr.group(store1, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.create_group('bar')
>>> baz = bar.create_dataset('baz', shape=100, chunks=50, dtype='i8')
>>> import numpy as np
>>> baz[:] = np.arange(100)
>>> root.tree()
/
 └── foo
     └── bar
         └── baz (100,) int64
>>> from sys import stdout
>>> store2 = zarr.ZipStore('data/example.zip', mode='w')
>>> zarr.copy_store(store1, store2, log=stdout)
copy .zgroup
copy foo/.zgroup
copy foo/bar/.zgroup
copy foo/bar/baz/.zarray
copy foo/bar/baz/0
copy foo/bar/baz/1
all done: 6 copied, 0 skipped, 566 bytes copied
(6, 0, 566)
>>> new_root = zarr.group(store2)
>>> new_root.tree()
/
 └── foo
     └── bar
         └── baz (100,) int64
>>> new_root['foo/bar/baz'][:]
array([ 0,  1,  2,  ..., 97, 98, 99])
>>> store2.close()  # zip stores need to be closed
```

zarr.convenience.**tree**(*grp*, *expand=False*, *level=None*)

Provide a `print`-able display of the hierarchy. This function is provided mainly as a convenience for obtaining a tree view of an h5py group - zarr groups have a `.tree()` method.

Parameters

grp
[Group] Zarr or h5py group.

expand
[bool, optional] Only relevant for HTML representation. If True, tree will be fully expanded.

level
[int, optional] Maximum depth to descend into hierarchy.

See also:

zarr.hierarchy.Group.tree

Notes

Please note that this is an experimental feature. The behaviour of this function is still evolving and the default output and/or parameters may change in future versions.

Examples

```
>>> import zarr
>>> g1 = zarr.group()
>>> g2 = g1.create_group('foo')
>>> g3 = g1.create_group('bar')
>>> g4 = g3.create_group('baz')
>>> g5 = g3.create_group('qux')
>>> d1 = g5.create_dataset('baz', shape=100, chunks=10)
>>> g1.tree()
/
 ├── bar
 │   ├── baz
 │   └── qux
 │       └── baz (100,) float64
 └── foo
>>> import h5py
>>> h5f = h5py.File('data/example.h5', mode='w')
>>> zarr.copy_all(g1, h5f)
(5, 0, 800)
>>> zarr.tree(h5f)
/
 ├── bar
 │   ├── baz
 │   └── qux
 │       └── baz (100,) float64
 └── foo
```

zarr.convenience.**consolidate_metadata**(*store: BaseStore*, *metadata_key='.zmetadata'*, ***, *path=''*)

Consolidate all metadata for groups and arrays within the given store into a single resource and put it under the given key.

This produces a single object in the backend store, containing all the metadata read from all the zarr-related keys that can be found. After metadata have been consolidated, use *open_consolidated()* to open the root group in optimised, read-only mode, using the consolidated metadata to reduce the number of read operations on the backend store.

Note, that if the metadata in the store is changed after this consolidation, then the metadata read by *open_consolidated()* would be incorrect unless this function is called again.

Note: This is an experimental feature.

Parameters

store
[MutableMapping or string] Store or path to directory in file system or name of zip file.

metadata_key
[str] Key to put the consolidated metadata under.

path
[str or None] Path corresponding to the group that is being consolidated. Not required for zarr v2 stores.

Returns

g
[*zarr.hierarchy.Group*] Group instance, opened with the new consolidated metadata.

See also:

open_consolidated

zarr.convenience.**open_consolidated**(*store: BaseStore | MutableMapping | str | None*, *metadata_key='.zmetadata'*, *mode='r+'*, ***kwargs*)

Open group using metadata previously consolidated into a single key.

This is an optimised method for opening a Zarr group, where instead of traversing the group/array hierarchy by accessing the metadata keys at each level, a single key contains all of the metadata for everything. For remote data sources where the overhead of accessing a key is large compared to the time to read data.

The group accessed must have already had its metadata consolidated into a single key using the function *consolidate_metadata()*.

This optimised method only works in modes which do not change the metadata, although the data may still be written/updated.

Parameters

store
[MutableMapping or string] Store or path to directory in file system or name of zip file.

metadata_key
[str] Key to read the consolidated metadata from. The default (.zmetadata) corresponds to the default used by *consolidate_metadata()*.

mode
[{'r', 'r+'}, optional] Persistence mode: 'r' means read only (must exist); 'r+' means read/write (must exist) although only writes to data are allowed, changes to metadata including creation of new arrays or group are not allowed.

****kwargs**
Additional parameters are passed through to *zarr.creation.open_array()* or *zarr.hierarchy.open_group()*.

Returns

g
[*zarr.hierarchy.Group*] Group instance, opened with the consolidated metadata.

See also:

consolidate_metadata

3.7 Compressors and filters (zarr.codecs)

This module contains compressor and filter classes for use with Zarr. Please note that this module is provided for backwards compatibility with previous versions of Zarr. From Zarr version 2.2 onwards, all codec classes have been moved to a separate package called Numcodecs. The two packages (Zarr and Numcodecs) are designed to be used together. For example, a Numcodecs codec class can be used as a compressor for a Zarr array:

```
>>> import zarr
>>> from numcodecs import Blosc
>>> z = zarr.zeros(1000000, compressor=Blosc(cname='zstd', clevel=1, shuffle=Blosc.
↪SHUFFLE))
```

Codec classes can also be used as filters. See the tutorial section on *Filters* for more information.

Please note that it is also relatively straightforward to define and register custom codec classes. See the Numcodecs codec API and codec registry documentation for more information.

3.8 The Attributes class (zarr.attrs)

class zarr.attrs.**Attributes**(*store*, *key='.zattrs'*, *read_only=False*, *cache=True*, *synchronizer=None*)

Class providing access to user attributes on an array or group. Should not be instantiated directly, will be available via the *.attrs* property of an array or group.

Parameters

store
[MutableMapping] The store in which to store the attributes.

key
[str, optional] The key under which the attributes will be stored.

read_only
[bool, optional] If True, attributes cannot be modified.

cache
[bool, optional] If True (default), attributes will be cached locally.

synchronizer
[Synchronizer] Only necessary if attributes may be modified from multiple threads or processes.

__getitem__(*item*)

__setitem__(*item*, *value*)

__delitem__(*item*)

__iter__()

__len__()

keys() → a set-like object providing a view on D's keys

asdict()

Retrieve all attributes as a dictionary.

put(*d*)

Overwrite all attributes with the key/value pairs in the provided dictionary *d* in a single operation.

update(**args*, ***kwargs*)

Update the values of several attributes in a single operation.

refresh()

Refresh cached attributes from the store.

3.9 Synchronization (zarr.sync)

class zarr.sync.**ThreadSynchronizer**

Provides synchronization using thread locks.

class zarr.sync.**ProcessSynchronizer**(*path*)

Provides synchronization using file locks via the fasteners package.

Parameters

path

[string] Path to a directory on a file system that is shared by all processes. N.B., this should be a *different* path to where you store the array.

3.10 V3 Specification Implementation(zarr._storage.v3)

This module contains the implementation of the Zarr V3 Specification.

Warning: Since Zarr Python 2.12 release, this module provides experimental infrastructure for reading and writing the upcoming V3 spec of the Zarr format. Users wishing to prepare for the migration can set the environment variable `ZARR_V3_EXPERIMENTAL_API=1` to begin experimenting, however data written with this API should be expected to become stale, as the implementation will still change.

The new `zarr._store.v3` package has the necessary classes and functions for evaluating Zarr V3. Since the design is not finalised, the classes and functions are not automatically imported into the regular Zarr namespace.

Code snippet for creating Zarr V3 arrays:

```
>>> import zarr
>>> z = zarr.create((10000, 10000),
>>>             chunks=(100, 100),
>>>             dtype='f8',
>>>             compressor='default',
>>>             path='path-where-you-want-zarr-v3-array',
>>>             zarr_version=3)
```

Further, you can use *z.info* to see details about the array you just created:

```
>>> z.info
Name               : path-where-you-want-zarr-v3-array
Type               : zarr.core.Array
Data type          : float64
```

(continues on next page)

(continued from previous page)

```
Shape               : (10000, 10000)
Chunk shape         : (100, 100)
Order               : C
Read-only           : False
Compressor          : Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type          : zarr._storage.v3.KVStoreV3
No. bytes           : 800000000 (762.9M)
No. bytes stored    : 557
Storage ratio       : 1436265.7
Chunks initialized : 0/10000
```

You can also check `Store type` here (which indicates Zarr V3).

class zarr._storage.v3.**RmdirV3**

Mixin class that can be used to ensure override of any existing v2 rmdir class.

class zarr._storage.v3.**KVStoreV3**(*mutablemapping*)

This provides a default implementation of a store interface around a mutable mapping, to avoid having to test stores for presence of methods.

This, for most methods should just be a pass-through to the underlying KV store which is likely to expose a MuttableMapping interface,

class zarr._storage.v3.**FSStoreV3**(*url*, *normalize_keys=False*, *key_separator=None*, *mode='w'*, *exceptions=(<class 'KeyError'>, <class 'PermissionError'>, <class 'OSError'>)*, *dimension_separator: ~typing.Literal['.', '/'] | None = None*, *fs=None*, *check=False*, *create=False*, *missing_exceptions=None*, ***storage_options*)

class zarr._storage.v3.**MemoryStoreV3**(*root=None*, *cls=<class 'dict'>*, *dimension_separator: ~typing.Literal['.', '/'] | None = None*)

Store class that uses a hierarchy of `KVStore` objects, thus all data will be held in main memory.

Notes

Safe to write in multiple threads.

Examples

This is the default class used when creating a group. E.g.:

```
>>> import zarr
>>> g = zarr.group()
>>> type(g.store)
<class 'zarr.storage.MemoryStore'>
```

Note that the default class when creating an array is the built-in `KVStore` class, i.e.:

```
>>> z = zarr.zeros(100)
>>> type(z.store)
<class 'zarr.storage.KVStore'>
```

class zarr._storage.v3.**DirectoryStoreV3**(*path*, *normalize_keys=False*, *dimension_separator: Literal['.', '/'] | None = None*)

Storage class using directories and files on a standard file system.

Parameters

path
[string] Location of directory to use as the root of the storage hierarchy.

normalize_keys
[bool, optional] If True, all store keys will be normalized to use lower case characters (e.g. 'foo' and 'FOO' will be treated as equivalent). This can be useful to avoid potential discrepancies between case-sensitive and case-insensitive file system. Default value is False.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

Notes

Atomic writes are used, which means that data are first written to a temporary file, then moved into place when the write is successfully completed. Files are only held open while they are being read or written and are closed immediately afterwards, so there is no need to manually close any files.

Safe to write in multiple threads or processes.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.DirectoryStore('data/array.zarr')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
```

Each chunk of the array is stored as a separate file on the file system, i.e.:

```
>>> import os
>>> sorted(os.listdir('data/array.zarr'))
['.zarray', '0.0', '0.1', '1.0', '1.1']
```

Store a group:

```
>>> store = zarr.DirectoryStore('data/group.zarr')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
```

When storing a group, levels in the group hierarchy will correspond to directories on the file system, i.e.:

```
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup', 'foo']
>>> sorted(os.listdir('data/group.zarr/foo'))
['.zgroup', 'bar']
```

(continues on next page)

(continued from previous page)

```
>>> sorted(os.listdir('data/group.zarr/foo/bar'))
['.zarray', '0.0', '0.1', '1.0', '1.1']
```

class zarr._storage.v3.**ZipStoreV3**(*path*, *compression=0*, *allowZip64=True*, *mode='a'*, *dimension_separator: Literal['.', '/'] | None = None*)

Storage class using a Zip file.

Parameters

path
[string] Location of file.

compression
[integer, optional] Compression method to use when writing to the archive.

allowZip64
[bool, optional] If True (the default) will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 2 GiB. If False will raise an exception when the ZIP file would require ZIP64 extensions.

mode
[string, optional] One of 'r' to read an existing file, 'w' to truncate and write a new file, 'a' to append to an existing file, or 'x' to exclusively create and write a new file.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

Notes

Each chunk of an array is stored as a separate entry in the Zip file. Note that Zip files do not provide any way to remove or replace existing entries. If an attempt is made to replace an entry, then a warning is generated by the Python standard library about a duplicate Zip file entry. This can be triggered if you attempt to write data to a Zarr array more than once, e.g.:

```
>>> store = zarr.ZipStore('data/example.zip', mode='w')
>>> z = zarr.zeros(100, chunks=10, store=store)
>>> # first write OK
... z[...] = 42
>>> # second write generates warnings
... z[...] = 42
>>> store.close()
```

This can also happen in a more subtle situation, where data are written only once to a Zarr array, but the write operations are not aligned with chunk boundaries, e.g.:

```
>>> store = zarr.ZipStore('data/example.zip', mode='w')
>>> z = zarr.zeros(100, chunks=10, store=store)
>>> z[5:15] = 42
>>> # write overlaps chunk previously written, generates warnings
... z[15:25] = 42
```

To avoid creating duplicate entries, only write data once, and align writes with chunk boundaries. This alignment is done automatically if you call `z[...] = ...` or create an array from existing data via `zarr.array()`.

Alternatively, use a `DirectoryStore` when writing the data, then manually Zip the directory and use the Zip file for subsequent reads. Take note that the files in the Zip file must be relative to the root of the Zarr archive. You may find it easier to create such a Zip file with `7z`, e.g.:

```
7z a -tzip archive.zarr.zip archive.zarr/.
```

Safe to write in multiple threads but not in multiple processes.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.ZipStore('data/array.zip', mode='w')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.ZipStore('data/group.zip', mode='w')
>>> root = zarr.group(store=store)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

After modifying a ZipStore, the `close()` method must be called, otherwise essential data will not be written to the underlying Zip file. The ZipStore class also supports the context manager protocol, which ensures the `close()` method is called on leaving the context, e.g.:

```
>>> with zarr.ZipStore('data/array.zip', mode='w') as store:
...     z = zarr.zeros((10, 10), chunks=(5, 5), store=store)
...     z[...] = 42
...     # no need to call store.close()
```

class zarr._storage.v3.**RedisStoreV3**(*prefix='zarr'*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using Redis.

Note: This is an experimental feature.

Requires the redis package to be installed.

Parameters

prefix
[string] Name of prefix for Redis keys

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *redis.Redis* function.

class zarr._storage.v3.**MongoDBStoreV3**(*database='mongodb_zarr'*, *collection='zarr_collection'*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using MongoDB.

Note: This is an experimental feature.

Requires the pymongo package to be installed.

Parameters

database
[string] Name of database

collection
[string] Name of collection

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *pymongo.MongoClient* function.

Notes

The maximum chunksize in MongoDB documents is 16 MB.

class zarr._storage.v3.**DBMStoreV3**(*path*, *flag='c'*, *mode=438*, *open=None*, *write_lock=True*, *dimension_separator: Literal['.', '/'] | None = None*, ***open_kwargs*)

Storage class using a DBM-style database.

Parameters

path
[string] Location of database file.

flag
[string, optional] Flags for opening the database file.

mode
[int] File mode used if a new file is created.

open
[function, optional] Function to open the database file. If not provided, `dbm.open()` will be used on Python 3, and `anydbm.open()` will be used on Python 2.

write_lock: bool, optional
Use a lock to prevent concurrent writes from multiple threads (True by default).

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.e

****open_kwargs**
Keyword arguments to pass the *open* function.

Notes

Please note that, by default, this class will use the Python standard library *dbm.open* function to open the database file (or *anydbm.open* on Python 2). There are up to three different implementations of DBM-style databases available in any Python installation, and which one is used may vary from one system to another. Database file formats are not compatible between these different implementations. Also, some implementations are more efficient than others. In particular, the "dumb" implementation will be the fall-back on many systems, and has very poor performance for some usage scenarios. If you want to ensure a specific implementation is used, pass the corresponding open function, e.g., *dbm.gnu.open* to use the GNU DBM library.

Safe to write in multiple threads. May be safe to write in multiple processes, depending on which DBM implementation is being used, although this has not been tested.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.DBMStore('data/array.db')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.DBMStore('data/group.db')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

After modifying a DBMStore, the `close()` method must be called, otherwise essential data may not be written to the underlying database file. The DBMStore class also supports the context manager protocol, which ensures the `close()` method is called on leaving the context, e.g.:

```
>>> with zarr.DBMStore('data/array.db') as store:
...     z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
...     z[...] = 42
...     # no need to call store.close()
```

A different database library can be used by passing a different function to the *open* parameter. For example, if the bsddb3 package is installed, a Berkeley DB database can be used:

```
>>> import bsddb3
>>> store = zarr.DBMStore('data/array.bdb', open=bsddb3.btopen)
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()
```

class zarr._storage.v3.**LMDBStoreV3**(*path*, *buffers=True*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using LMDB. Requires the lmdb package to be installed.

Parameters

path
[string] Location of database file.

buffers
[bool, optional] If True (default) use support for buffers, which should increase performance by reducing memory copies.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *lmdb.open* function.

Notes

By default writes are not immediately flushed to disk to increase performance. You can ensure data are flushed to disk by calling the `flush()` or `close()` methods.

Should be safe to write in multiple threads or processes due to the synchronization support within LMDB, although writing from multiple processes has not been tested.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.LMDBStore('data/array.mdb')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.LMDBStore('data/group.mdb')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

After modifying a DBMStore, the `close()` method must be called, otherwise essential data may not be written to the underlying database file. The DBMStore class also supports the context manager protocol, which ensures the `close()` method is called on leaving the context, e.g.:

```
>>> with zarr.LMDBStore('data/array.mdb') as store:
...     z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
...     z[...] = 42
...     # no need to call store.close()
```

class zarr._storage.v3.**SQLiteStoreV3**(*path*, *dimension_separator: Literal['.', '/'] | None = None*, ***kwargs*)

Storage class using SQLite.

Parameters

path
[string] Location of database file.

dimension_separator
[{'.', '/'}, optional] Separator placed between the dimensions of a chunk.

****kwargs**
Keyword arguments passed through to the *sqlite3.connect* function.

Examples

Store a single array:

```
>>> import zarr
>>> store = zarr.SQLiteStore('data/array.sqldb')
>>> z = zarr.zeros((10, 10), chunks=(5, 5), store=store, overwrite=True)
>>> z[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

Store a group:

```
>>> store = zarr.SQLiteStore('data/group.sqldb')
>>> root = zarr.group(store=store, overwrite=True)
>>> foo = root.create_group('foo')
>>> bar = foo.zeros('bar', shape=(10, 10), chunks=(5, 5))
>>> bar[...] = 42
>>> store.close()  # don't forget to call this when you're done
```

class zarr._storage.v3.**LRUStoreCacheV3**(*store*, *max_size: int*)

Storage class that implements a least-recently-used (LRU) cache layer over some other store. Intended primarily for use with stores that can be slow to access, e.g., remote stores that require network communication to store and retrieve data.

Parameters

store
[Store] The store containing the actual data to be cached.

max_size
[int] The maximum size that the cache may grow to, in number of bytes. Provide *None* if you would like the cache to have unlimited size.

Examples

The example below wraps an S3 store with an LRU cache:

```
>>> import s3fs
>>> import zarr
>>> s3 = s3fs.S3FileSystem(anon=True, client_kwargs=dict(region_name='eu-west-2'))
>>> store = s3fs.S3Map(root='zarr-demo/store', s3=s3, check=False)
>>> cache = zarr.LRUStoreCache(store, max_size=2**28)
>>> root = zarr.group(store=cache)
>>> z = root['foo/bar/baz']
>>> from timeit import timeit
>>> # first data access is relatively slow, retrieved from store
... timeit('print(z[:].tobytes())', number=1, globals=globals())
b'Hello from the cloud!'
```

(continues on next page)

(continued from previous page)

```
0.1081731989979744
>>> # second data access is faster, uses cache
... timeit('print(z[:].tobytes())', number=1, globals=globals())
b'Hello from the cloud!'
0.0009490990014455747
```

class zarr._storage.v3.**ConsolidatedMetadataStoreV3**(*store: BaseStore | MutableMapping*, *metadata_key='meta/root/consolidated/.zmetadata'*)

A layer over other storage, where the metadata has been consolidated into a single key.

The purpose of this class, is to be able to get all of the metadata for a given array in a single read operation from the underlying storage. See *zarr.convenience.consolidate_metadata()* for how to create this single metadata key.

This class loads from the one key, and stores the data in a dict, so that accessing the keys no longer requires operations on the backend store.

This class is read-only, and attempts to change the array metadata will fail, but changing the data is possible. If the backend storage is changed directly, then the metadata stored here could become obsolete, and *zarr.convenience.consolidate_metadata()* should be called again and the class re-invoked. The use case is for write once, read many times.

Note: This is an experimental feature.

Parameters

store: Store
Containing the zarr array.

metadata_key: str
The target in the store where all of the metadata are stored. We assume JSON encoding.

See also:

zarr.convenience.consolidate_metadata, *zarr.convenience.open_consolidated*

In v3 storage transformers can be set via `zarr.create(..., storage_transformers=[...])`. The experimental sharding storage transformer can be tested by setting the environment variable `ZARR_V3_SHARDING=1`. Data written with this flag enabled should be expected to become stale until ZEP 2 is approved and fully implemented.

class zarr._storage.v3_storage_transformers.**ShardingStorageTransformer**(*_type*, *chunks_per_shard*)

Implements sharding as a storage transformer, as described in the spec: https://zarr-specs.readthedocs.io/en/latest/extensions/storage-transformers/sharding/v1.0.html https://purl.org/zarr/spec/storage_transformers/sharding/1.0

The abstract base class for storage transformers is

class zarr._storage.store.**StorageTransformer**(*_type*)

Base class for storage transformers. The methods simply pass on the data as-is and should be overwritten by sub-classes.

3.11 Indices and tables

- genindex
- modindex
- search

CHAPTER

FOUR

SPECIFICATIONS

4.1 Zarr Storage Specification Version 3

The V3 Specification has been migrated to its website → https://zarr-specs.readthedocs.io/.

4.2 Zarr Storage Specification Version 2

This document provides a technical specification of the protocol and format used for storing Zarr arrays. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

4.2.1 Status

This specification is the latest version. See *Specifications* for previous versions.

4.2.2 Storage

A Zarr array can be stored in any storage system that provides a key/value interface, where a key is an ASCII string and a value is an arbitrary sequence of bytes, and the supported operations are read (get the sequence of bytes associated with a given key), write (set the sequence of bytes associated with a given key) and delete (remove a key/value pair).

For example, a directory in a file system can provide this interface, where keys are file names, values are file contents, and files can be read, written or deleted via the operating system. Equally, an S3 bucket can provide this interface, where keys are resource names, values are resource contents, and resources can be read, written or deleted via HTTP.

Below an "array store" refers to any system implementing this interface.

4.2.3 Arrays

Metadata

Each array requires essential configuration metadata to be stored, enabling correct interpretation of the stored data. This metadata is encoded using JSON and stored as the value of the ".zarray" key within an array store.

The metadata resource is a JSON object. The following keys MUST be present within the object:

zarr_format
: An integer defining the version of the storage specification to which the array store adheres.

shape
A list of integers defining the length of each dimension of the array.

chunks
A list of integers defining the length of each dimension of a chunk of the array. Note that all chunks within a Zarr array have the same shape.

dtype
A string or list defining a valid data type for the array. See also the subsection below on data type encoding.

compressor
A JSON object identifying the primary compression codec and providing configuration parameters, or `null` if no compressor is to be used. The object MUST contain an `"id"` key identifying the codec to be used.

fill_value
A scalar value providing the default value to use for uninitialized portions of the array, or `null` if no fill_value is to be used.

order
Either "C" or "F", defining the layout of bytes within each chunk of the array. "C" means row-major order, i.e., the last dimension varies fastest; "F" means column-major order, i.e., the first dimension varies fastest.

filters
A list of JSON objects providing codec configurations, or `null` if no filters are to be applied. Each codec configuration object MUST contain a `"id"` key identifying the codec to be used.

The following keys MAY be present within the object:

dimension_separator
If present, either the string `"."` or `"/"` defining the separator placed between the dimensions of a chunk. If the value is not set, then the default MUST be assumed to be `"."`, leading to chunk keys of the form "0.0". Arrays defined with `"/"` as the dimension separator can be considered to have nested, or hierarchical, keys of the form "0/0" that SHOULD where possible produce a directory-like structure.

Other keys SHOULD NOT be present within the metadata object and SHOULD be ignored by implementations.

For example, the JSON object below defines a 2-dimensional array of 64-bit little-endian floating point numbers with 10000 rows and 10000 columns, divided into chunks of 1000 rows and 1000 columns (so there will be 100 chunks in total arranged in a 10 by 10 grid). Within each chunk the data are laid out in C contiguous order. Each chunk is encoded using a delta filter and compressed using the Blosc compression library prior to storage:

```
{
    "chunks": [
        1000,
        1000
    ],
    "compressor": {
        "id": "blosc",
        "cname": "lz4",
        "clevel": 5,
        "shuffle": 1
    },
    "dtype": "<f8",
    "fill_value": "NaN",
    "filters": [
        {"id": "delta", "dtype": "<f8", "astype": "<f4"}
    ],
    "order": "C",
```

(continues on next page)

(continued from previous page)

```
    "shape": [
        10000,
        10000
    ],
    "zarr_format": 2
}
```

Data type encoding

Simple data types are encoded within the array metadata as a string, following the NumPy array protocol type string (typestr) format. The format consists of 3 parts:

- One character describing the byteorder of the data ("<": little-endian; ">": big-endian; "|": not-relevant)
- One character code giving the basic type of the array ("b": Boolean (integer type where all values are only True or False); "i": integer; "u": unsigned integer; "f": floating point; "c": complex floating point; "m": timedelta; "M": datetime; "S": string (fixed-length sequence of char); "U": unicode (fixed-length sequence of Py_UNICODE); "V": other (void * – each item is a fixed-size chunk of memory))
- An integer specifying the number of bytes the type uses.

The byte order MUST be specified. E.g., "<f8", ">i4", "|b1" and "|S12" are valid data type encodings.

For datetime64 ("M") and timedelta64 ("m") data types, these MUST also include the units within square brackets. A list of valid units and their definitions are given in the NumPy documentation on Datetimes and Timedeltas. For example, "<M8[ns]" specifies a datetime64 data type with nanosecond time units.

Structured data types (i.e., with multiple named fields) are encoded as a list of lists, following NumPy array protocol type descriptions (descr). Each sub-list has the form [fieldname, datatype, shape] where shape is optional. fieldname is a string, datatype is a string specifying a simple data type (see above), and shape is a list of integers specifying subarray shape. For example, the JSON list below defines a data type composed of three single-byte unsigned integer fields named "r", "g" and "b":

```
[["r", "|u1"], ["g", "|u1"], ["b", "|u1"]]
```

For example, the JSON list below defines a data type composed of three fields named "x", "y" and "z", where "x" and "y" each contain 32-bit floats, and each item in "z" is a 2 by 2 array of floats:

```
[["x", "<f4"], ["y", "<f4"], ["z", "<f4", [2, 2]]]
```

Structured data types may also be nested, e.g., the following JSON list defines a data type with two fields "foo" and "bar", where "bar" has two sub-fields "baz" and "qux":

```
[["foo", "<f4"], ["bar", [["baz", "<f4"], ["qux", "<i4"]]]]
```

Fill value encoding

For simple floating point data types, the following table MUST be used to encode values of the "fill_value" field:

Value	JSON encoding
Not a Number	`"NaN"`
Positive Infinity	`"Infinity"`
Negative Infinity	`"-Infinity"`

If an array has a fixed length byte string data type (e.g., `"|S12"`), or a structured data type, and if the fill value is not null, then the fill value MUST be encoded as an ASCII string using the standard Base64 alphabet.

Chunks

Each chunk of the array is compressed by passing the raw bytes for the chunk through the primary compression library to obtain a new sequence of bytes comprising the compressed chunk data. No header is added to the compressed bytes or any other modification made. The internal structure of the compressed bytes will depend on which primary compressor was used. For example, the Blosc compressor produces a sequence of bytes that begins with a 16-byte header followed by compressed data.

The compressed sequence of bytes for each chunk is stored under a key formed from the index of the chunk within the grid of chunks representing the array. To form a string key for a chunk, the indices are converted to strings and concatenated with the period character (".") separating each index. For example, given an array with shape (10000, 10000) and chunk shape (1000, 1000) there will be 100 chunks laid out in a 10 by 10 grid. The chunk with indices (0, 0) provides data for rows 0-999 and columns 0-999 and is stored under the key "0.0"; the chunk with indices (2, 4) provides data for rows 2000-2999 and columns 4000-4999 and is stored under the key "2.4"; etc.

There is no need for all chunks to be present within an array store. If a chunk is not present then it is considered to be in an uninitialized state. An uninitialized chunk MUST be treated as if it was uniformly filled with the value of the "fill_value" field in the array metadata. If the "fill_value" field is `null` then the contents of the chunk are undefined.

Note that all chunks in an array have the same shape. If the length of any array dimension is not exactly divisible by the length of the corresponding chunk dimension then some chunks will overhang the edge of the array. The contents of any chunk region falling outside the array are undefined.

Filters

Optionally a sequence of one or more filters can be used to transform chunk data prior to compression. When storing data, filters are applied in the order specified in array metadata to encode data, then the encoded data are passed to the primary compressor. When retrieving data, stored chunk data are decompressed by the primary compressor then decoded using filters in the reverse order.

4.2.4 Hierarchies

Logical storage paths

Multiple arrays can be stored in the same array store by associating each array with a different logical path. A logical path is simply an ASCII string. The logical path is used to form a prefix for keys used by the array. For example, if an array is stored at logical path "foo/bar" then the array metadata will be stored under the key "foo/bar/.zarray", the user-defined attributes will be stored under the key "foo/bar/.zattrs", and the chunks will be stored under keys like "foo/bar/0.0", "foo/bar/0.1", etc.

To ensure consistent behaviour across different storage systems, logical paths MUST be normalized as follows:

- Replace all backward slash characters ("\\") with forward slash characters ("/")
- Strip any leading "/" characters
- Strip any trailing "/" characters
- Collapse any sequence of more than one "/" character into a single "/" character

The key prefix is then obtained by appending a single "/" character to the normalized logical path.

After normalization, if splitting a logical path by the "/" character results in any path segment equal to the string "." or the string ".." then an error MUST be raised.

N.B., how the underlying array store processes requests to store values under keys containing the "/" character is entirely up to the store implementation and is not constrained by this specification. E.g., an array store could simply treat all keys as opaque ASCII strings; equally, an array store could map logical paths onto some kind of hierarchical storage (e.g., directories on a file system).

Groups

Arrays can be organized into groups which can also contain other groups. A group is created by storing group metadata under the ".zgroup" key under some logical path. E.g., a group exists at the root of an array store if the ".zgroup" key exists in the store, and a group exists at logical path "foo/bar" if the "foo/bar/.zgroup" key exists in the store.

If the user requests a group to be created under some logical path, then groups MUST also be created at all ancestor paths. E.g., if the user requests group creation at path "foo/bar" then groups MUST be created at path "foo" and the root of the store, if they don't already exist.

If the user requests an array to be created under some logical path, then groups MUST also be created at all ancestor paths. E.g., if the user requests array creation at path "foo/bar/baz" then groups must be created at path "foo/bar", path "foo", and the root of the store, if they don't already exist.

The group metadata resource is a JSON object. The following keys MUST be present within the object:

zarr_format
: An integer defining the version of the storage specification to which the array store adheres.

Other keys MUST NOT be present within the metadata object.

The members of a group are arrays and groups stored under logical paths that are direct children of the parent group's logical path. E.g., if groups exist under the logical paths "foo" and "foo/bar" and an array exists at logical path "foo/baz" then the members of the group at path "foo" are the group at path "foo/bar" and the array at path "foo/baz".

4.2.5 Attributes

An array or group can be associated with custom attributes, which are arbitrary key/value pairs with application-specific meaning. Custom attributes are encoded as a JSON object and stored under the ".zattrs" key within an array store. The ".zattrs" key does not have to be present, and if it is absent the attributes should be treated as empty.

For example, the JSON object below encodes three attributes named "foo", "bar" and "baz":

```
{
    "foo": 42,
    "bar": "apples",
    "baz": [1, 2, 3, 4]
}
```

4.2.6 Examples

Storing a single array

Below is an example of storing a Zarr array, using a directory on the local file system as storage.

Create an array:

```
>>> import zarr
>>> store = zarr.DirectoryStore('data/example.zarr')
>>> a = zarr.create(shape=(20, 20), chunks=(10, 10), dtype='i4',
...                 fill_value=42, compressor=zarr.Zlib(level=1),
...                 store=store, overwrite=True)
```

No chunks are initialized yet, so only the ".zarray" and ".zattrs" keys have been set in the store:

```
>>> import os
>>> sorted(os.listdir('data/example.zarr'))
['.zarray']
```

Inspect the array metadata:

```
>>> print(open('data/example.zarr/.zarray').read())
{
    "chunks": [
        10,
        10
    ],
    "compressor": {
        "id": "zlib",
        "level": 1
    },
    "dtype": "<i4",
    "fill_value": 42,
    "filters": null,
    "order": "C",
    "shape": [
        20,
        20
    ],
    "zarr_format": 2
}
```

Chunks are initialized on demand. E.g., set some data:

```
>>> a[0:10, 0:10] = 1
>>> sorted(os.listdir('data/example.zarr'))
['.zarray', '0.0']
```

Set some more data:

```
>>> a[0:10, 10:20] = 2
>>> a[10:20, :] = 3
>>> sorted(os.listdir('data/example.zarr'))
['.zarray', '0.0', '0.1', '1.0', '1.1']
```

Manually decompress a single chunk for illustration:

```
>>> import zlib
>>> buf = zlib.decompress(open('data/example.zarr/0.0', 'rb').read())
>>> import numpy as np
>>> chunk = np.frombuffer(buf, dtype='<i4')
>>> chunk
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
```

Modify the array attributes:

```
>>> a.attrs['foo'] = 42
>>> a.attrs['bar'] = 'apples'
>>> a.attrs['baz'] = [1, 2, 3, 4]
>>> sorted(os.listdir('data/example.zarr'))
['.zarray', '.zattrs', '0.0', '0.1', '1.0', '1.1']
>>> print(open('data/example.zarr/.zattrs').read())
{
    "bar": "apples",
    "baz": [
        1,
        2,
        3,
        4
    ],
    "foo": 42
}
```

Storing multiple arrays in a hierarchy

Below is an example of storing multiple Zarr arrays organized into a group hierarchy, using a directory on the local file system as storage. This storage implementation maps logical paths onto directory paths on the file system, however this is an implementation choice and is not required.

Setup the store:

```
>>> import zarr
>>> store = zarr.DirectoryStore('data/group.zarr')
```

Create the root group:

```
>>> root_grp = zarr.group(store, overwrite=True)
```

The metadata resource for the root group has been created:

```
>>> import os
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup']
```

Inspect the group metadata:

```
>>> print(open('data/group.zarr/.zgroup').read())
{
    "zarr_format": 2
}
```

Create a sub-group:

```
>>> sub_grp = root_grp.create_group('foo')
```

What has been stored:

```
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup', 'foo']
>>> sorted(os.listdir('data/group.zarr/foo'))
['.zgroup']
```

Create an array within the sub-group:

```
>>> a = sub_grp.create_dataset('bar', shape=(20, 20), chunks=(10, 10))
>>> a[:] = 42
```

Set a custom attributes:

```
>>> a.attrs['comment'] = 'answer to life, the universe and everything'
```

What has been stored:

```
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup', 'foo']
>>> sorted(os.listdir('data/group.zarr/foo'))
['.zgroup', 'bar']
>>> sorted(os.listdir('data/group.zarr/foo/bar'))
['.zarray', '.zattrs', '0.0', '0.1', '1.0', '1.1']
```

Here is the same example using a Zip file as storage:

```
>>> store = zarr.ZipStore('data/group.zip', mode='w')
>>> root_grp = zarr.group(store)
>>> sub_grp = root_grp.create_group('foo')
>>> a = sub_grp.create_dataset('bar', shape=(20, 20), chunks=(10, 10))
>>> a[:] = 42
>>> a.attrs['comment'] = 'answer to life, the universe and everything'
>>> store.close()
```

What has been stored:

```
>>> import zipfile
>>> zf = zipfile.ZipFile('data/group.zip', mode='r')
>>> for name in sorted(zf.namelist()):
...     print(name)
.zgroup
foo/.zgroup
foo/bar/.zarray
foo/bar/.zattrs
```

(continues on next page)

(continued from previous page)

```
foo/bar/0.0
foo/bar/0.1
foo/bar/1.0
foo/bar/1.1
```

4.2.7 Changes

Version 2 clarifications

The following changes have been made to the version 2 specification since it was initially published to clarify ambiguities and add some missing information.

- The specification now describes how bytes fill values should be encoded and decoded for arrays with a fixed-length byte string data type (#165, #176).
- The specification now clarifies that units must be specified for datetime64 and timedelta64 data types (#85, #215).
- The specification now clarifies that the '.zattrs' key does not have to be present for either arrays or groups, and if absent then custom attributes should be treated as empty.
- The specification now describes how structured datatypes with subarray shapes and/or with nested structured data types are encoded in array metadata (#111, #296).
- Clarified the key/value pairs of custom attributes as "arbitrary" rather than "simple".

Changes from version 1 to version 2

The following changes were made between version 1 and version 2 of this specification:

- Added support for storing multiple arrays in the same store and organising arrays into hierarchies using groups.
- Array metadata is now stored under the ".zarray" key instead of the "meta" key.
- Custom attributes are now stored under the ".zattrs" key instead of the "attrs" key.
- Added support for filters.
- Changed encoding of "fill_value" field within array metadata.
- Changed encoding of compressor information within array metadata to be consistent with representation of filter information.

4.3 Zarr Storage Specification Version 1

This document provides a technical specification of the protocol and format used for storing a Zarr array. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

4.3.1 Status

This specification is deprecated. See *Specifications* for the latest version.

4.3.2 Storage

A Zarr array can be stored in any storage system that provides a key/value interface, where a key is an ASCII string and a value is an arbitrary sequence of bytes, and the supported operations are read (get the sequence of bytes associated with a given key), write (set the sequence of bytes associated with a given key) and delete (remove a key/value pair).

For example, a directory in a file system can provide this interface, where keys are file names, values are file contents, and files can be read, written or deleted via the operating system. Equally, an S3 bucket can provide this interface, where keys are resource names, values are resource contents, and resources can be read, written or deleted via HTTP.

Below an "array store" refers to any system implementing this interface.

4.3.3 Metadata

Each array requires essential configuration metadata to be stored, enabling correct interpretation of the stored data. This metadata is encoded using JSON and stored as the value of the 'meta' key within an array store.

The metadata resource is a JSON object. The following keys MUST be present within the object:

zarr_format
: An integer defining the version of the storage specification to which the array store adheres.

shape
: A list of integers defining the length of each dimension of the array.

chunks
: A list of integers defining the length of each dimension of a chunk of the array. Note that all chunks within a Zarr array have the same shape.

dtype
: A string or list defining a valid data type for the array. See also the subsection below on data type encoding.

compression
: A string identifying the primary compression library used to compress each chunk of the array.

compression_opts
: An integer, string or dictionary providing options to the primary compression library.

fill_value
: A scalar value providing the default value to use for uninitialized portions of the array.

order
: Either 'C' or 'F', defining the layout of bytes within each chunk of the array. 'C' means row-major order, i.e., the last dimension varies fastest; 'F' means column-major order, i.e., the first dimension varies fastest.

Other keys MAY be present within the metadata object however they MUST NOT alter the interpretation of the required fields defined above.

For example, the JSON object below defines a 2-dimensional array of 64-bit little-endian floating point numbers with 10000 rows and 10000 columns, divided into chunks of 1000 rows and 1000 columns (so there will be 100 chunks in total arranged in a 10 by 10 grid). Within each chunk the data are laid out in C contiguous order, and each chunk is compressed using the Blosc compression library:

```
{
    "chunks": [
        1000,
        1000
    ],
    "compression": "blosc",
    "compression_opts": {
        "clevel": 5,
        "cname": "lz4",
        "shuffle": 1
    },
    "dtype": "<f8",
    "fill_value": null,
    "order": "C",
    "shape": [
        10000,
        10000
    ],
    "zarr_format": 1
}
```

Data type encoding

Simple data types are encoded within the array metadata resource as a string, following the NumPy array protocol type string (typestr) format. The format consists of 3 parts: a character describing the byteorder of the data (<: little-endian, >: big-endian, |: not-relevant), a character code giving the basic type of the array, and an integer providing the number of bytes the type uses. The byte order MUST be specified. E.g., "<f8", ">i4", "|b1" and "|S12" are valid data types.

Structure data types (i.e., with multiple named fields) are encoded as a list of two-element lists, following NumPy array protocol type descriptions (descr). For example, the JSON list `[["r", "|u1"], ["g", "|u1"], ["b", "|u1"]]` defines a data type composed of three single-byte unsigned integers labelled 'r', 'g' and 'b'.

4.3.4 Chunks

Each chunk of the array is compressed by passing the raw bytes for the chunk through the primary compression library to obtain a new sequence of bytes comprising the compressed chunk data. No header is added to the compressed bytes or any other modification made. The internal structure of the compressed bytes will depend on which primary compressor was used. For example, the Blosc compressor produces a sequence of bytes that begins with a 16-byte header followed by compressed data.

The compressed sequence of bytes for each chunk is stored under a key formed from the index of the chunk within the grid of chunks representing the array. To form a string key for a chunk, the indices are converted to strings and concatenated with the period character ('.') separating each index. For example, given an array with shape (10000, 10000) and chunk shape (1000, 1000) there will be 100 chunks laid out in a 10 by 10 grid. The chunk with indices (0, 0) provides data for rows 0-999 and columns 0-999 and is stored under the key '0.0'; the chunk with indices (2, 4) provides data for rows 2000-2999 and columns 4000-4999 and is stored under the key '2.4'; etc.

There is no need for all chunks to be present within an array store. If a chunk is not present then it is considered to be in an uninitialized state. An uninitialized chunk MUST be treated as if it was uniformly filled with the value of the 'fill_value' field in the array metadata. If the 'fill_value' field is `null` then the contents of the chunk are undefined.

Note that all chunks in an array have the same shape. If the length of any array dimension is not exactly divisible by the length of the corresponding chunk dimension then some chunks will overhang the edge of the array. The contents of any chunk region falling outside the array are undefined.

4.3.5 Attributes

Each array can also be associated with custom attributes, which are simple key/value items with application-specific meaning. Custom attributes are encoded as a JSON object and stored under the 'attrs' key within an array store. Even if the attributes are empty, the 'attrs' key MUST be present within an array store.

For example, the JSON object below encodes three attributes named 'foo', 'bar' and 'baz':

```
{
    "foo": 42,
    "bar": "apples",
    "baz": [1, 2, 3, 4]
}
```

4.3.6 Example

Below is an example of storing a Zarr array, using a directory on the local file system as storage.

Initialize the store:

```
>>> import zarr
>>> store = zarr.DirectoryStore('example.zarr')
>>> zarr.init_store(store, shape=(20, 20), chunks=(10, 10),
...                 dtype='i4', fill_value=42, compression='zlib',
...                 compression_opts=1, overwrite=True)
```

No chunks are initialized yet, so only the 'meta' and 'attrs' keys have been set:

```
>>> import os
>>> sorted(os.listdir('example.zarr'))
['attrs', 'meta']
```

Inspect the array metadata:

```
>>> print(open('example.zarr/meta').read())
{
    "chunks": [
        10,
        10
    ],
    "compression": "zlib",
    "compression_opts": 1,
    "dtype": "<i4",
    "fill_value": 42,
    "order": "C",
    "shape": [
        20,
        20
    ],
```

(continues on next page)

(continued from previous page)

```
    "zarr_format": 1
}
```

Inspect the array attributes:

```
>>> print(open('example.zarr/attrs').read())
{}
```

Set some data:

```
>>> z = zarr.Array(store)
>>> z[0:10, 0:10] = 1
>>> sorted(os.listdir('example.zarr'))
['0.0', 'attrs', 'meta']
```

Set some more data:

```
>>> z[0:10, 10:20] = 2
>>> z[10:20, :] = 3
>>> sorted(os.listdir('example.zarr'))
['0.0', '0.1', '1.0', '1.1', 'attrs', 'meta']
```

Manually decompress a single chunk for illustration:

```
>>> import zlib
>>> b = zlib.decompress(open('example.zarr/0.0', 'rb').read())
>>> import numpy as np
>>> a = np.frombuffer(b, dtype='<i4')
>>> a
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
```

Modify the array attributes:

```
>>> z.attrs['foo'] = 42
>>> z.attrs['bar'] = 'apples'
>>> z.attrs['baz'] = [1, 2, 3, 4]
>>> print(open('example.zarr/attrs').read())
{
    "bar": "apples",
    "baz": [
        1,
        2,
        3,
        4
    ],
    "foo": 42
}
```

CHAPTER

FIVE

RELEASE NOTES

5.1 Unreleased

5.2 2.17.1

5.2.1 Enhancements

- Change occurrences of % and format() to f-strings. By Dimitri Papadopoulos Orfanos #1423.
- Proper argument for numpy.reshape. By Dimitri Papadopoulos Orfanos #1425.
- Add typing to dimension separator arguments. By David Stansby #1620.

5.2.2 Docs

- ZIP related tweaks. By Davis Bennett #1641.

5.2.3 Maintenance

- Update config.yml with Zulip. By Josh Moore.
- Replace Gitter with the new Zulip Chat link. By Sanket Verma #1685.
- Fix RTD build. By Sanket Verma #1694.

5.3 2.17.0

5.3.1 Enhancements

- Added type hints to `zarr.creation.create()`. By David Stansby #1536.
- Pyodide support: Don't require fasteners on Emscripten. By Hood Chatham #1663.

5.3.2 Docs

- Minor correction and changes in documentation. By Sanket Verma #1509.
- Fix typo in documentation. By Dimitri Papadopoulos Orfanos #1554
- The documentation build now fails if there are any warnings. By David Stansby #1548.
- Add links to `numcodecs` docs in the tutorial. By David Stansby #1535.
- Enable offline formats for documentation builds. By Sanket Verma #1551.
- Minor tweak to advanced indexing tutorial examples. By Ross Barnowski #1550.
- Automatically document array members using sphinx-automodapi. By David Stansby #1547.
- Add a markdown file documenting the current and former core-developer team. By Joe Hamman #1628.
- Add Norman Rzepka to core-dev team. By Joe Hamman #1630.
- Added section about accessing ZIP archives on s3. By Jeff Peck #1613, #1615, and Davis Bennett #1641.
- Add V3 roadmap and design document. By Joe Hamman #1583.

5.3.3 Maintenance

- Drop Python 3.8 and NumPy 1.20 By Josh Moore; #1557.
- Cache result of `FSStore._fsspec_installed()`. By Janick Martinez Esturo #1581.
- Extend copyright notice to 2023. By Jack Kelly #1528.
- Change occurrence of `io.open()` into `open()`. By Dimitri Papadopoulos Orfanos #1421.
- Preserve `dimension_separator` when resizing arrays. By Ziwen Liu #1533.
- Initialise some sets in tests with set literals instead of list literals. By Dimitri Papadopoulos Orfanos #1534.
- Allow `black` code formatter to be run with any Python version. By David Stansby #1549.
- Remove `sphinx-rtd-theme` dependency from `pyproject.toml`. By Sanket Verma #1563.
- Remove `CODE_OF_CONDUCT.md` file from the Zarr-Python repository. By Sanket Verma #1572.
- Bump version of black in pre-commit. By David Stansby #1559.
- Use list comprehension where applicable. By Dimitri Papadopoulos Orfanos #1555.
- Use format specification mini-language to format string. By Dimitri Papadopoulos Orfanos #1558.
- Single startswith() call instead of multiple ones. By Dimitri Papadopoulos Orfanos #1556.
- Move codespell options around. By Dimitri Papadopoulos Orfanos #1196.
- Remove unused mypy ignore comments. By David Stansby #1602.

5.4 2.16.1

5.4.1 Maintenance

- Require `setuptools_scm` version `1.5.4+` By John A. Kirkham #1477.
- Add `docs` requirements to `pyproject.toml` By John A. Kirkham #1494.
- Fixed caching issue in `LRUStoreCache`. By Mads R. B. Kristensen #1499.

5.5 2.16.0

5.5.1 Enhancements

- Allow for partial codec specification in V3 array metadata. By Joe Hamman #1443.
- Add `__contains__` method to `KVStore`. By Christoph Gohlke #1454.
- **Block Indexing**: Implemented blockwise (chunk blocks) indexing to `zarr.Array`. By Altay Sansal #1428

5.5.2 Maintenance

- Refactor the core array tests to reduce code duplication. By Davis Bennett #1462.
- Style the codebase with `ruff` and `black`. By Davis Bennett #1459
- Ensure that chunks is tuple of ints upon array creation. By Philipp Hanslovsky #1461

5.6 2.15.0

5.6.1 Enhancements

- Implement more extensive fallback of getitem/setitem for orthogonal indexing. By Andreas Albert #1029.
- Getitems supports `meta_array`. By Mads R. B. Kristensen #1131.
- `open_array()` now takes the `meta_array` argument. By Mads R. B. Kristensen #1396.

5.6.2 Maintenance

- Remove `codecov` from GitHub actions. By John A. Kirkham #1391.
- Replace `np.product` with `np.prod` due to deprecation. By James Bourbeau #1405.
- Activate Py 3.11 builds. By Joe Hamman #1415.

5.6.3 Documentation

- Add API reference for V3 Implementation in the docs. By Sanket Verma #1345.

5.6.4 Bug fixes

- Fix the conda-forge error. Read #1347 for detailed info. By Josh Moore #1364 and #1367.
- Fix `ReadOnlyError` when opening V3 store via fsspec reference file system. By Joe Hamman #1383.
- Fix `normalize_fill_value` for structured arrays. By Alan Du #1397.

5.7 2.14.2

5.7.1 Bug fixes

- Ensure `zarr.group` uses writeable mode to fix issue with #1304. By Brandur Thorgrimsson #1354.

5.8 2.14.1

5.8.1 Documentation

- Fix API links. By Josh Moore #1346.
- Fix unit tests which prevented the conda-forge release. By Josh Moore #1348.

5.9 2.14.0

5.9.1 Major changes

- Improve Zarr V3 support, adding partial store read/write and storage transformers. Add new features from the v3 spec:
  - storage transformers
  - *get_partial_values* and *set_partial_values*
  - efficient *get_partial_values* implementation for *FSStoreV3*
  - sharding storage transformer

  By Jonathan Striebel; #1096, #1111.
- N5 nows supports Blosc. Remove warnings emitted when using N5Store or N5FSStore with a blosc-compressed array. By Davis Bennett; #1331.

5.9.2 Bug fixes

- Allow reading utf-8 encoded json files By Nathan Zimmerberg #1308.
- Ensure contiguous data is give to `FSStore`. Only copying if needed. By Mads R. B. Kristensen #1285.
- NestedDirectoryStore.listdir now returns chunk keys with the correct '/' dimension_separator. By Brett Graham #1334.
- N5Store/N5FSStore dtype returns zarr Stores readable dtype. By Marwan Zouinkhi #1339.

5.10 2.13.6

5.10.1 Maintenance

- Bump gh-action-pypi-publish to 1.6.4. By Josh Moore #1320.

5.11 2.13.5

5.11.1 Bug fixes

- Ensure `zarr.create` uses writeable mode to fix issue with #1304. By James Bourbeau #1309.

5.12 2.13.4

5.12.1 Appreciation

Special thanks to Outreachy participants for contributing to most of the maintenance PRs. Please read the blog post summarising the contribution phase and welcoming new Outreachy interns: https://zarr.dev/blog/welcoming-outreachy-2022-interns/

5.12.2 Enhancements

- Handle fsspec.FSMap using FSStore store. By Rafal Wojdyla #1304.

5.12.3 Bug fixes

- Fix bug that caused double counting of groups in `groups()` and `group_keys()` methods with V3 stores. By Ryan Abernathey #1228.
- Remove unnecessary calling of *contains_array* for key that ended in *.array.json*. By Joe Hamman #1149.
- Fix bug that caused double counting of groups in `groups()` and `group_keys()` methods with V3 stores. By Ryan Abernathey #1228.

5.12.4 Documentation

- Fix minor indexing errors in tutorial and specification examples of documentation. By Kola Babalola #1277.
- Add *requirements_rtfd.txt* in *contributing.rst*. By AWA BRANDON AWA #1243.
- Add documentation for find/findall using visit. By Weddy Gikunda #1241.
- Refresh of the main landing page. By Josh Moore #1173.

5.12.5 Maintenance

- Migrate to `pyproject.toml` and remove redundant infrastructure. By Saransh Chopra #1158.
- Require `setuptools` 64.0.0+ By Saransh Chopra #1193.
- Pin action versions (pypi-publish, setup-miniconda) for dependabot By Saransh Chopra #1205.
- Remove `tox` support By Saransh Chopra #1219.
- Add workflow to label PRs with "needs release notes". By Saransh Chopra #1239.
- Simplify if/else statement. By Dimitri Papadopoulos Orfanos #1227.
- Get coverage up to 100%. By John Kirkham #1264.
- Migrate coverage to `pyproject.toml`. By John Kirkham #1250.
- Use `conda-incubator/setup-miniconda@v2.2.0`. By John Kirkham #1263.
- Delete unused files. By John Kirkham #1251.
- Skip labeller for bot PRs. By Saransh Chopra #1271.
- Restore Flake8 configuration. By John Kirkham #1249.
- Add missing newline at EOF. By @Dimitri Papadopoulos #1253.
- Add *license_files* to *pyproject.toml*. By John Kirkham #1247.
- Adding *pyupgrade* suggestions. By Dimitri Papadopoulos Orfanos #1225.
- Fixed some linting errors. By Weddy Gikunda #1226.
- Added the link to main website in readthedocs sidebar. By Stephanie_nkwatoh #1216.
- Remove redundant wheel dependency in *pyproject.toml*. By Dimitri Papadopoulos Orfanos #1233.
- Turned on *isloated_build* in *tox.ini* file. By AWA BRANDON AWA #1210.
- Fixed *flake8* alert and avoid duplication of *Zarr Developers*. By Dimitri Papadopoulos Orfanos #1203.
- Bump to NumPy 1.20+ in *environment.yml*. By John Kirkham #1201.
- Bump to NumPy 1.20 in *pyproject.toml*. By Dimitri Papadopoulos Orfanos #1192.
- Remove LGTM (*.lgtm.yml*) configuration file. By Dimitri Papadopoulos Orfanos #1191.
- Codespell will skip *fixture* in pre-commit. By Dimitri Papadopoulos Orfanos #1197.
- Add msgpack in *requirements_rtfd.txt*. By Emmanuel Bolarinwa #1188.
- Added license to docs fixed a typo from *_spec_v2* to *_spec_v3*. By AWA BRANDON AWA #1182.
- Fixed installation link in *README.md*. By AWA BRANDON AWA #1177.
- Fixed typos in *installation.rst* and *release.rst*. By Chizoba Nweke #1178.

- Set *docs/conf.py* language to *en*. By AWA BRANDON AWA #1174.
- Added *installation.rst* to the docs. By AWA BRANDON AWA #1170.
- Adjustment of year to *2015-2018* to *2015-2022* in the docs. By Emmanuel Bolarinwa #1165.
- Updated *Forking the repository* section in *contributing.rst*. By AWA BRANDON AWA #1171.
- Updated GitHub actions. By Dimitri Papadopoulos Orfanos #1134.
- Update web links: *http://* → *https://*. By Dimitri Papadopoulos Orfanos #1313.

5.13 2.13.3

- Improve performance of slice selections with steps by omitting chunks with no relevant data. By Richard Shaw #843.

5.14 2.13.2

- Fix test failure on conda-forge builds (again). By Josh Moore; see zarr-feedstock#65.

5.15 2.13.1

- Fix test failure on conda-forge builds. By Josh Moore; see zarr-feedstock#65.

5.16 2.13.0

5.16.1 Major changes

- **Support of alternative array classes** by introducing a new argument, meta_array, that specifies the type/class of the underlying array. The meta_array argument can be any class instance that can be used as the like argument in NumPy (see NEP 35). enabling support for CuPy through, for example, the creation of a CuPy CPU compressor. By Mads R. B. Kristensen #934.
- **Remove support for Python 3.7** in concert with NumPy dependency. By Davis Bennett #1067.
- **Zarr v3: add support for the default root path** rather than requiring that all API users pass an explicit path. By Gregory R. Lee #1085, #1142.

5.16.2 Bug fixes

- Remove/relax erroneous "meta" path check (**regression**). By Gregory R. Lee #1123.
- Cast all attribute keys to strings (and issue deprecation warning). By Mattia Almansi #1066.
- Fix bug in N5 storage that prevented arrays located in the root of the hierarchy from bearing the *n5* keyword. Along with fixing this bug, new tests were added for N5 routines that had previously been excluded from testing, and type annotations were added to the N5 codebase. By Davis Bennett #1092.
- Fix bug in LRUEStoreCache in which the current size wasn't reset on invalidation. By BGCMHou and Josh Moore #1076, #1077.

- Remove erroneous check that disallowed array keys starting with "meta". By Gregory R. Lee #1105.

5.16.3 Documentation

- Typo fixes to close quotes. By Pavithra Eswaramoorthy
- Added copy button to documentation. By Altay Sansal #1124.

5.16.4 Maintenance

- Simplify release docs. By Josh Moore #1119.
- Pin werkzeug to prevent test hangs. By Davis Bennett #1098.
- Fix a few DeepSource.io alerts By Dimitri Papadopoulos Orfanos #1080.
- Fix URLs. By Dimitri Papadopoulos Orfanos, #1074.
- Fix spelling. By Dimitri Papadopoulos Orfanos, #1073.
- Update GitHub issue templates with *YAML* format. By Saransh Chopra #1079.
- Remove option to return None from _ensure_store. By Gregory Lee #1068.
- Fix a typo of "integers". By Richard Scott #1056.

5.17 2.12.0

5.17.1 Enhancements

- **Add support for reading and writing Zarr V3.** The new *zarr._store.v3* package has the necessary classes and functions for evaluating Zarr V3. Since the format is not yet finalized, the classes and functions are not automatically imported into the regular *zarr* name space. Setting the *ZARR_V3_EXPERIMENTAL_API* environment variable will activate them. By Gregory Lee; #898, #1006, and #1007 as well as by Josh Moore #1032.
- **Create FSStore from an existing fsspec filesystem**. If you have created an fsspec filesystem outside of Zarr, you can now pass it as a keyword argument to `FSStore`. By Ryan Abernathey; #911.
- Add numpy encoder class for json.dumps By Eric Prestat; #933.
- Appending performance improvement to Zarr arrays, e.g., when writing to S3. By hailiangzhang; #1014.
- Add number encoder for `json.dumps` to support numpy integers in `chunks` arguments. By Eric Prestat #697.

5.17.2 Bug fixes

- Fix bug that made it impossible to create an `FSStore` on unlistable filesystems (e.g. some HTTP servers). By Ryan Abernathey; #993.

5.17.3 Documentation

- Update resize doc to clarify surprising behavior. By hailiangzhang; #1022.

5.17.4 Maintenance

- Added Pre-commit configuration, incl. Yaml Check. By Shivank Chaudhary; #1015, #1016.
- Fix URL to renamed file in Blosc repo. By Andrew Thomas #1028.
- Activate Py 3.10 builds. By Josh Moore #1027.
- Make all unignored zarr warnings errors. By Josh Moore #1021.

5.18 2.11.3

5.18.1 Bug fixes

- Fix missing case to fully revert change to default write_empty_chunks. By Tom White; #1005.

5.19 2.11.2

5.19.1 Bug fixes

- Changes the default value of `write_empty_chunks` to `True` to prevent unanticipated data losses when the data types do not have a proper default value when empty chunks are read back in. By Vyas Ramasubramani; #965, #1001.

5.20 2.11.1

5.20.1 Bug fixes

- Fix bug where indexing with a scalar numpy value returned a single-value array. By Ben Jeffery #967.
- Removed *clobber* argument from *normalize_store_arg*. This enables to change data within an opened consolidated group using mode *"r+"* (i.e region write). By Tobias Kölling #975.

5.21 2.11.0

5.21.1 Enhancements

- **Sparse changes with performance impact!** One of the advantages of the Zarr format is that it is sparse, which means that chunks with no data (more precisely, with data equal to the fill value, which is usually 0) don't need to be written to disk at all. They will simply be assumed to be empty at read time. However, until this release, the Zarr library would write these empty chunks to disk anyway. This changes in this version: a small performance penalty at write time leads to significant speedups at read time and in filesystem operations in the case of sparse

arrays. To revert to the old behavior, pass the argument `write_empty_chunks=True` to the array creation function. By Juan Nunez-Iglesias; #853 and Davis Bennett; #738.

- **Fancy indexing**. Zarr arrays now support NumPy-style fancy indexing with arrays of integer coordinates. This is equivalent to using zarr.Array.vindex. Mixing slices and integer arrays is not supported. By Juan Nunez-Iglesias; #725.
- **New base class**. This release of Zarr Python introduces a new `BaseStore` class that all provided store classes implemented in Zarr Python now inherit from. This is done as part of refactoring to enable future support of the Zarr version 3 spec. Existing third-party stores that are a MutableMapping (e.g. dict) can be converted to a new-style key/value store inheriting from `BaseStore` by passing them as the argument to the new `zarr.storage.KVStore` class. For backwards compatibility, various higher-level array creation and convenience functions still accept plain Python dicts or other mutable mappings for the `store` argument, but will internally convert these to a `KVStore`. By Gregory Lee; #839, #789, and #950.
- Allow to assign array `fill_values` and update metadata accordingly. By Ryan Abernathey, #662.
- Allow to update array fill_values By Matthias Bussonnier #665.

5.21.2 Bug fixes

- Fix bug where the checksum of zipfiles is wrong By Oren Watson #930.
- Fix consolidate_metadata with FSStore. By Joe Hamman #916.
- Unguarded next inside generator. By Dimitri Papadopoulos Orfanos #889.

5.21.3 Documentation

- Update docs creation of dev env. By Ray Bell #921.
- Update docs to use `python -m pytest`. By Ray Bell #923.
- Fix versionadded tag in zarr.core.Array docstring. By Juan Nunez-Iglesias #852.
- Doctest seem to be stricter now, updating tostring() to tobytes(). By John Kirkham #907.
- Minor doc fix. By Mads R. B. Kristensen #937.

5.21.4 Maintenance

- Upgrade MongoDB in test env. By Joe Hamman #939.
- Pass dimension_separator on fixture generation. By Josh Moore #858.
- Activate Python 3.9 in GitHub Actions. By Josh Moore #859.
- Drop shortcut `fsspec[s3]` for dependency. By Josh Moore #920.
- and a swath of code-linting improvements by Dimitri Papadopoulos Orfanos:
  - Unnecessary comprehension (#899)
  - Unnecessary `None` provided as default (#900)
  - use an if `expression` instead of *and/or* (#888)
  - Remove unnecessary literal (#891)
  - Decorate a few method with *@staticmethod* (#885)

- Drop unneeded `return` (#884)
- Drop explicit `object` inheritance from `class`-es (#886)
- Unnecessary comprehension (#883)
- Codespell configuration (#882)
- Fix typos found by codespell (#880)
- Proper C-style formatting for integer (#913)
- Add LGTM.com / DeepSource.io configuration files (#909)

5.22 2.10.3

5.22.1 Bug fixes

- N5 keywords now emit UserWarning instead of raising a ValueError. By Boaz Mohar; #860.
- blocks_to_decompress not used in read_part function. By Boaz Mohar; #861.
- defines blocksize for array, updates hexdigest values. By Andrew Fulton; #867.
- Fix test failure on Debian and conda-forge builds. By Josh Moore; #871.

5.23 2.10.2

5.23.1 Bug fixes

- Fix NestedDirectoryStore datasets without dimension_separator metadata. By Josh Moore; #850.

5.24 2.10.1

5.24.1 Bug fixes

- Fix regression by setting normalize_keys=False in fsstore constructor. By Davis Bennett; #842.

5.25 2.10.0

5.25.1 Enhancements

- Add N5FSStore. By Davis Bennett; #793.

5.25.2 Bug fixes

- Ignore None dim_separators in save_array. By Josh Moore; #831.

5.26 2.9.5

5.26.1 Bug fixes

- Fix FSStore.listdir behavior for nested directories. By Gregory Lee; #802.

5.27 2.9.4

5.27.1 Bug fixes

- Fix structured arrays that contain objects By :user: *Attila Bergou <abergou>*; :issue: *806*

5.28 2.9.3

5.28.1 Maintenance

- Mark the fact that some tests that require `fsspec`, without compromising the code coverage score. By Ben Williams; #823.
- Only inspect alternate node type if desired isn't present. By Trevor Manz; #696.

5.29 2.9.2

5.29.1 Maintenance

- Correct conda-forge deployment of Zarr by fixing some Zarr tests. By Ben Williams; #821.

5.30 2.9.1

5.30.1 Maintenance

- Correct conda-forge deployment of Zarr. By Josh Moore; #819.

5.31 2.9.0

This release of Zarr Python is the first release of Zarr to not support Python 3.6.

5.31.1 Enhancements

- Update ABSStore for compatibility with newer *azure.storage.blob*. By Tom Augspurger; #759.
- Pathlib support. By Chris Barnes; #768.

5.31.2 Documentation

- Clarify that arbitrary key/value pairs are OK for attributes. By Stephan Hoyer; #751.
- Clarify how to manually convert a DirectoryStore to a ZipStore. By pmav99; #763.

5.31.3 Bug fixes

- Fix dimension_separator support. By Josh Moore; #775.
- Extract ABSStore to zarr._storage.absstore. By Josh Moore; #781.
- avoid NumPy 1.21.0 due to https://github.com/numpy/numpy/issues/19325 By Gregory Lee; #791.

5.31.4 Maintenance

- Drop 3.6 builds. By Josh Moore; #774, #778.
- Fix build with Sphinx 4. By Elliott Sales de Andrade; #799.
- TST: add missing assert in test_hexdigest. By Gregory Lee; #801.

5.32 2.8.3

5.32.1 Bug fixes

- FSStore: default to normalize_keys=False By Josh Moore; #755.
- ABSStore: compatibility with `azure.storage.python`>=12 By Tom Augspurger; #618

5.33 2.8.2

5.33.1 Documentation

- Add section on rechunking to tutorial By David Baddeley; #730.

5.33.2 Bug fixes

- Expand FSStore tests and fix implementation issues By Davis Bennett; #709.

5.33.3 Maintenance

- Updated ipytree warning for jlab3 By Ian Hunt-Isaak; #721.
- b170a48a - (issue-728, copy-nested) Updated ipytree warning for jlab3 (#721) (3 weeks ago) <Ian Hunt-Isaak>
- Activate dependabot By Josh Moore; #734.
- Update Python classifiers (Zarr is stable!) By Josh Moore; #731.

5.34 2.8.1

5.34.1 Bug fixes

- raise an error if create_dataset's dimension_separator is inconsistent By Gregory R. Lee; #724.

5.35 2.8.0

5.35.1 V2 Specification Update

- Introduce optional dimension_separator .zarray key for nested chunks. By Josh Moore; #715, #716.

5.36 2.7.1

5.36.1 Bug fixes

- Update Array to respect FSStore's key_separator (#718) By Gregory R. Lee; #718.

5.37 2.7.0

5.37.1 Enhancements

- Start stop for iterator (*islice()*) By Sebastian Grill; #621.
- Add capability to partially read and decompress chunks By Andrew Fulton; #667.

5.37.2 Bug fixes

- Make DirectoryStore __setitem__ resilient against antivirus file locking By Eric Younkin; #698.
- Compare test data's content generally By John Kirkham; #436.
- Fix dtype usage in zarr/meta.py By Josh Moore; #700.
- Fix FSStore key_seperator usage By Josh Moore; #669.
- Simplify text handling in DB Store By John Kirkham; #670.
- GitHub Actions migration By Matthias Bussonnier; #641, #671, #674, #676, #677, #678, #679, #680, #682, #684, #685, #686, #687, #695, #706.

5.38 2.6.1

- Minor build fix By Matthias Bussonnier; #666.

5.39 2.6.0

This release of Zarr Python is the first release of Zarr to not support Python 3.5.

- End Python 3.5 support. By Chris Barnes; #602.
- Fix `open_group/open_array` to allow opening of read-only store with `mode='r'` #269
- Add *Array* tests for FSStore. By Andrew Fulton; :issue: *644*.
- fix a bug in which `attrs` would not be copied on the root when using `copy_all`; #613
- Fix `FileNotFoundError` with dask/s3fs #649
- Fix flaky fixture in test_storage.py #652
- Fix FSStore getitems fails with arrays that have a 0 length shape dimension #644
- Use async to fetch/write result concurrently when possible. #536, See this comment for some performance analysis showing order of magnitude faster response in some benchmark.

See this link for the full list of closed and merged PR tagged with the 2.6 milestone.

- Add ability to partially read and decompress arrays, see #667. It is only available to chunks stored using fsspec and using Blosc as a compressor.

  For certain analysis case when only a small portion of chunks is needed it can be advantageous to only access and decompress part of the chunks. Doing partial read and decompression add high latency to many of the operation so should be used only when the subset of the data is small compared to the full chunks and is stored contiguously (that is to say either last dimensions for C layout, firsts for F). Pass `partial_decompress=True` as argument when creating an `Array`, or when using `open_array`. No option exists yet to apply partial read and decompress on a per-operation basis.

5.40 2.5.0

This release will be the last to support Python 3.5, next version of Zarr will be Python 3.6+.

- *DirectoryStore* now uses *os.scandir*, which should make listing large store faster, #563
- Remove a few remaining Python 2-isms. By Poruri Sai Rahul; #393.
- Fix minor bug in *N5Store*. By @gsakkis, #550.
- Improve error message in Jupyter when trying to use the `ipytree` widget without `ipytree` installed. By Zain Patel; #537
- Add typing information to many of the core functions #589
- Explicitly close stores during testing. By Elliott Sales de Andrade; #442
- Many of the convenience functions to emit errors (`err_*` from `zarr.errors` have been replaced by `ValueError` subclasses. The corresponding `err_*` function have been removed. #590, #614)
- Improve consistency of terminology regarding arrays and datasets in the documentation. By Josh Moore; #571.
- Added support for generic URL opening by `fsspec`, where the URLs have the form "protocol://[server]/path" or can be chained URls with "::" separators. The additional argument `storage_options` is passed to the backend, see the `fsspec` docs. By Martin Durant; #546
- Added support for fetching multiple items via `getitems` method of a store, if it exists. This allows for concurrent fetching of data blocks from stores that implement this; presently HTTP, S3, GCS. Currently only applies to reading. By Martin Durant; #606
- Efficient iteration expanded with option to pass start and stop index via `array.islice`. By Sebastian Grill, #615.

5.41 2.4.0

5.41.1 Enhancements

- Add key normalization option for `DirectoryStore`, `NestedDirectoryStore`, `TempStore`, and `N5Store`. By James Bourbeau; #459.
- Add `recurse` keyword to `Group.array_keys` and `Group.arrays` methods. By James Bourbeau; #458.
- Use uniform chunking for all dimensions when specifying `chunks` as an integer. Also adds support for specifying `-1` to chunk across an entire dimension. By James Bourbeau; #456.
- Rename `DictStore` to `MemoryStore`. By James Bourbeau; #455.
- Rewrite `.tree()` pretty representation to use `ipytree`. Allows it to work in both the Jupyter Notebook and JupyterLab. By John Kirkham; #450.
- Do not rename Blosc parameters in n5 backend and add *blocksize* parameter, compatible with n5-blosc. By @axtimwalde, #485.
- Update `DirectoryStore` to create files with more permissive permissions. By Eduardo Gonzalez and James Bourbeau; #493
- Use `math.ceil` for scalars. By John Kirkham; #500.
- Ensure contiguous data using `astype`. By John Kirkham; #513.
- Refactor out `_tofile`/`_fromfile` from `DirectoryStore`. By John Kirkham; #503.

- Add `__enter__`/`__exit__` methods to `Group` for `h5py.File` compatibility. By Chris Barnes; #509.

5.41.2 Bug fixes

- Fix Sqlite Store Wrong Modification. By Tommy Tran; #440.
- Add intermediate step (using `zipfile.ZipInfo` object) to write inside `ZipStore` to solve too restrictive permission issue. By Raphael Dussin; #505.
- Fix '/' prepend bug in `ABSStore`. By Shikhar Goenka; #525.

5.41.3 Documentation

- Fix hyperlink in `README.md`. By Anderson Banihirwe; #531.
- Replace "nuimber" with "number". By John Kirkham; #512.
- Fix azure link rendering in tutorial. By James Bourbeau; #507.
- Update `README` file to be more detailed. By Zain Patel; #495.
- Import blosc from numcodecs in tutorial. By James Bourbeau; #491.
- Adds logo to docs. By James Bourbeau; #462.
- Fix N5 link in tutorial. By James Bourbeau; #480.
- Fix typo in code snippet. By Joe Jevnik; #461.
- Fix URLs to point to zarr-python By John Kirkham; #453.

5.41.4 Maintenance

- Add documentation build to CI. By James Bourbeau; #516.
- Use `ensure_ndarray` in a few more places. By John Kirkham; #506.
- Support Python 3.8. By John Kirkham; #499.
- Require Numcodecs 0.6.4+ to use text handling functionality from it. By John Kirkham; #497.
- Updates tests to use `pytest.importorskip`. By James Bourbeau; #492
- Removed support for Python 2. By @jhamman; #393, #470.
- Upgrade dependencies in the test matrices and resolve a compatibility issue with testing against the Azure Storage Emulator. By @alimanfoo; #468, #467.
- Use `unittest.mock` on Python 3. By Elliott Sales de Andrade; #426.
- Drop `decode` from `ConsolidatedMetadataStore`. By John Kirkham; #452.

5.42 2.3.2

5.42.1 Enhancements

- Use `scandir` in `DirectoryStore`'s `getsize` method. By John Kirkham; #431.

5.42.2 Bug fixes

- Add and use utility functions to simplify reading and writing JSON. By John Kirkham; #429, #430.
- Fix `collections`'s `DeprecationWarnings`. By John Kirkham; #432.
- Fix tests on big endian machines. By Elliott Sales de Andrade; #427.

5.43 2.3.1

5.43.1 Bug fixes

- Makes `azure-storage-blob` optional for testing. By John Kirkham; #419, #420.

5.44 2.3.0

5.44.1 Enhancements

- New storage backend, backed by Azure Blob Storage, class *zarr.storage.ABSStore*. All data is stored as block blobs. By Shikhar Goenka, Tim Crone and Zain Patel; #345.
- Add "consolidated" metadata as an experimental feature: use *zarr.convenience.consolidate_metadata()* to copy all metadata from the various metadata keys within a dataset hierarchy under a single key, and *zarr.convenience.open_consolidated()* to use this single key. This can greatly cut down the number of calls to the storage backend, and so remove a lot of overhead for reading remote data. By Martin Durant, Alistair Miles, Ryan Abernathey, #268, #332, #338.
- Support has been added for structured arrays with sub-array shape and/or nested fields. By Tarik Onalan, #111, #296.
- Adds the SQLite-backed *zarr.storage.SQLiteStore* class enabling an SQLite database to be used as the backing store for an array or group. By John Kirkham, #368, #365.
- Efficient iteration over arrays by decompressing chunkwise. By Jerome Kelleher, #398, #399.
- Adds the Redis-backed *zarr.storage.RedisStore* class enabling a Redis database to be used as the backing store for an array or group. By Joe Hamman, #299, #372.
- Adds the MongoDB-backed *zarr.storage.MongoDBStore* class enabling a MongoDB database to be used as the backing store for an array or group. By Noah D Brenowitz, Joe Hamman, #299, #372, #401.
- **New storage class for N5 containers**. The *zarr.n5.N5Store* has been added, which uses *zarr.storage.NestedDirectoryStore* to support reading and writing from and to N5 containers. By Jan Funke and John Kirkham.

5.44.2 Bug fixes

- The implementation of the *zarr.storage.DirectoryStore* class has been modified to ensure that writes are atomic and there are no race conditions where a chunk might appear transiently missing during a write operation. By sbalmer, #327, #263.
- Avoid raising in *zarr.storage.DirectoryStore*'s `__setitem__` when file already exists. By Justin Swaney, #272, #318.
- The required version of the Numcodecs package has been upgraded to 0.6.2, which has enabled some code simplification and fixes a failing test involving msgpack encoding. By John Kirkham, #361, #360, #352, #355, #324.
- Failing tests related to pickling/unpickling have been fixed. By Ryan Williams, #273, #308.
- Corrects handling of `NaT` in `datetime64` and `timedelta64` in various compressors (by John Kirkham; #344).
- Ensure `DictStore` contains only `bytes` to facilitate comparisons and protect against writes. By John Kirkham, #350.
- Test and fix an issue (w.r.t. fill values) when storing complex data to `Array`. By John Kirkham, #363.
- Always use a `tuple` when indexing a NumPy `ndarray`. By John Kirkham, #376.
- Ensure when `Array` uses a `dict`-based chunk store that it only contains `bytes` to facilitate comparisons and protect against writes. Drop the copy for the no filter/compressor case as this handles that case. By John Kirkham, #359.

5.44.3 Maintenance

- Simplify directory creation and removal in `DirectoryStore.rename`. By John Kirkham, #249.
- CI and test environments have been upgraded to include Python 3.7, drop Python 3.4, and upgrade all pinned package requirements. Alistair Miles, #308.
- Start using pyup.io to maintain dependencies. Alistair Miles, #326.
- Configure flake8 line limit generally. John Kirkham, #335.
- Add missing coverage pragmas. John Kirkham, #343, #355.
- Fix missing backslash in docs. John Kirkham, #254, #353.
- Include tests for stores' `popitem` and `pop` methods. By John Kirkham, #378, #380.
- Include tests for different compressors, endianness, and attributes. By John Kirkham, #378, #380.
- Test validity of stores' contents. By John Kirkham, #359, #408.

5.45 2.2.0

5.45.1 Enhancements

- **Advanced indexing**. The `Array` class has several new methods and properties that enable a selection of items in an array to be retrieved or updated. See the *Advanced indexing* tutorial section for more information. There is also a notebook with extended examples and performance benchmarks. #78, #89, #112, #172.

- **New package for compressor and filter codecs**. The classes previously defined in the *zarr.codecs* module have been factored out into a separate package called Numcodecs. The Numcodecs package also includes several new codec classes not previously available in Zarr, including compressor codecs for Zstd and LZ4. This change is backwards-compatible with existing code, as all codec classes defined by Numcodecs are imported into the *zarr.codecs* namespace. However, it is recommended to import codecs from the new package, see the tutorial sections on *Compressors* and *Filters* for examples. With contributions by John Kirkham; #74, #102, #120, #123, #139.
- **New storage class for DBM-style databases**. The *zarr.storage.DBMStore* class enables any DBM-style database such as gdbm, ndbm or Berkeley DB, to be used as the backing store for an array or group. See the tutorial section on *Storage alternatives* for some examples. #133, #186.
- **New storage class for LMDB databases**. The *zarr.storage.LMDBStore* class enables an LMDB "Lightning" database to be used as the backing store for an array or group. #192.
- **New storage class using a nested directory structure for chunk files**. The *zarr.storage.NestedDirectoryStore* has been added, which is similar to the existing *zarr.storage.DirectoryStore* class but nests chunk files for multidimensional arrays into sub-directories. #155, #177.
- **New tree() method for printing hierarchies**. The `Group` class has a new *zarr.hierarchy.Group.tree()* method which enables a tree representation of a group hierarchy to be printed. Also provides an interactive tree representation when used within a Jupyter notebook. See the *Array and group diagnostics* tutorial section for examples. By John Kirkham; #82, #140, #184.
- **Visitor API**. The `Group` class now implements the h5py visitor API, see docs for the *zarr.hierarchy.Group.visit()*, *zarr.hierarchy.Group.visititems()* and *zarr.hierarchy.Group.visitvalues()* methods. By John Kirkham, #92, #122.
- **Viewing an array as a different dtype**. The `Array` class has a new *zarr.core.Array.astype()* method, which is a convenience that enables an array to be viewed as a different dtype. By John Kirkham, #94, #96.
- **New open(), save(), load() convenience functions**. The function *zarr.convenience.open()* provides a convenient way to open a persistent array or group, using either a `DirectoryStore` or `ZipStore` as the backing store. The functions *zarr.convenience.save()* and *zarr.convenience.load()* are also available and provide a convenient way to save an entire NumPy array to disk and load back into memory later. See the tutorial section *Persistent arrays* for examples. #104, #105, #141, #181.
- **IPython completions**. The `Group` class now implements `__dir__()` and `_ipython_key_completions_()` which enables tab-completion for group members to be used in any IPython interactive environment. #170.
- **New info property; changes to __repr__**. The `Group` and `Array` classes have a new `info` property which can be used to print diagnostic information, including compression ratio where available. See the tutorial section on *Array and group diagnostics* for examples. The string representation (`__repr__`) of these classes has been simplified to ensure it is cheap and quick to compute in all circumstances. #83, #115, #132, #148.
- **Chunk options**. When creating an array, `chunks=False` can be specified, which will result in an array with a single chunk only. Alternatively, `chunks=True` will trigger an automatic chunk shape guess. See *Chunk optimizations* for more on the `chunks` parameter. #106, #107, #183.
- **Zero-dimensional arrays** and are now supported; by Prakhar Goel, #154, #161.
- **Arrays with one or more zero-length dimensions** are now fully supported; by Prakhar Goel, #150, #154, #160.
- **The .zattrs key is now optional** and will now only be created when the first custom attribute is set; #121, #200.
- **New Group.move() method** supports moving a sub-group or array to a different location within the same hierarchy. By John Kirkham, #191, #193, #196.
- **ZipStore is now thread-safe**; #194, #192.
- **New Array.hexdigest() method** computes an `Array`'s hash with `hashlib`. By John Kirkham, #98, #203.

- **Improved support for object arrays**. In previous versions of Zarr, creating an array with `dtype=object` was possible but could under certain circumstances lead to unexpected errors and/or segmentation faults. To make it easier to properly configure an object array, a new `object_codec` parameter has been added to array creation functions. See the tutorial section on *Object arrays* for more information and examples. Also, runtime checks have been added in both Zarr and Numcodecs so that segmentation faults are no longer possible, even with a badly configured array. This API change is backwards compatible and previous code that created an object array and provided an object codec via the `filters` parameter will continue to work, however a warning will be raised to encourage use of the `object_codec` parameter. #208, #212.
- **Added support for datetime64 and timedelta64 data types**; #85, #215.
- **Array and group attributes are now cached by default** to improve performance with slow stores, e.g., stores accessing data via the network; #220, #218, #204.
- **New LRUStoreCache class**. The class *zarr.storage.LRUStoreCache* has been added and provides a means to locally cache data in memory from a store that may be slow, e.g., a store that retrieves data from a remote server via the network; #223.
- **New copy functions**. The new functions *zarr.convenience.copy()* and *zarr.convenience.copy_all()* provide a way to copy groups and/or arrays between HDF5 and Zarr, or between two Zarr groups. The *zarr.convenience.copy_store()* provides a more efficient way to copy data directly between two Zarr stores. #87, #113, #137, #217.

5.45.2 Bug fixes

- Fixed bug where `read_only` keyword argument was ignored when creating an array; #151, #179.
- Fixed bugs when using a `ZipStore` opened in 'w' mode; #158, #182.
- Fill values can now be provided for fixed-length string arrays; #165, #176.
- Fixed a bug where the number of chunks initialized could be counted incorrectly; #97, #174.
- Fixed a bug related to the use of an ellipsis (`...`) in indexing statements; #93, #168, #172.
- Fixed a bug preventing use of other integer types for indexing; #143, #147.

5.45.3 Documentation

- Some changes have been made to the *Zarr Storage Specification Version 2* document to clarify ambiguities and add some missing information. These changes do not break compatibility with any of the material as previously implemented, and so the changes have been made in-place in the document without incrementing the document version number. See the section on *Changes* in the specification document for more information.
- A new *Advanced indexing* section has been added to the tutorial.
- A new *String arrays* section has been added to the tutorial (#135, #175).
- The *Chunk optimizations* tutorial section has been reorganised and updated.
- The *Persistent arrays* and *Storage alternatives* tutorial sections have been updated with new examples (#100, #101, #103).
- A new tutorial section on *Pickle support* has been added (#91).
- A new tutorial section on *Datetimes and timedeltas* has been added.
- A new tutorial section on *Array and group diagnostics* has been added.

- The tutorial sections on *Parallel computing and synchronization* and *Configuring Blosc* have been updated to provide information about how to avoid program hangs when using the Blosc compressor with multiple processes (#199, #201).

5.45.4 Maintenance

- A data fixture has been included in the test suite to ensure data format compatibility is maintained; #83, #146.
- The test suite has been migrated from nosetests to pytest; #189, #225.
- Various continuous integration updates and improvements; #118, #124, #125, #126, #109, #114, #171.
- Bump numcodecs dependency to 0.5.3, completely remove nose dependency, #237.
- Fix compatibility issues with NumPy 1.14 regarding fill values for structured arrays, #222, #238, #239.

5.45.5 Acknowledgments

Code was contributed to this release by Alistair Miles, John Kirkham and Prakhar Goel.

Documentation was contributed to this release by Mamy Ratsimbazafy and Charles Noyes.

Thank you to John Kirkham, Stephan Hoyer, Francesc Alted, and Matthew Rocklin for code reviews and/or comments on pull requests.

5.46 2.1.4

- Resolved an issue where calling `hasattr` on a `Group` object erroneously returned a `KeyError`. By Vincent Schut; #88, #95.

5.47 2.1.3

- Resolved an issue with *zarr.creation.array()* where dtype was given as None (#80).

5.48 2.1.2

- Resolved an issue when no compression is used and chunks are stored in memory (#79).

5.49 2.1.1

Various minor improvements, including: `Group` objects support member access via dot notation (`__getattr__`); fixed metadata caching for `Array.shape` property and derivatives; added `Array.ndim` property; fixed `Array.__array__` method arguments; fixed bug in pickling `Array` state; fixed bug in pickling `ThreadSynchronizer`.

5.50 2.1.0

- Group objects now support member deletion via `del` statement (#65).
- Added *zarr.storage.TempStore* class for convenience to provide storage via a temporary directory (#59).
- Fixed performance issues with *zarr.storage.ZipStore* class (#66).
- The Blosc extension has been modified to return bytes instead of array objects from compress and decompress function calls. This should improve compatibility and also provides a small performance increase for compressing high compression ratio data (#55).
- Added `overwrite` keyword argument to array and group creation methods on the *zarr.hierarchy.Group* class (#71).
- Added `cache_metadata` keyword argument to array creation methods.
- The functions *zarr.creation.open_array()* and *zarr.hierarchy.open_group()* now accept any store as first argument (#56).

5.51 2.0.1

The bundled Blosc library has been upgraded to version 1.11.1.

5.52 2.0.0

5.52.1 Hierarchies

Support has been added for organizing arrays into hierarchies via groups. See the tutorial section on *Groups* and the *zarr.hierarchy* API docs for more information.

5.52.2 Filters

Support has been added for configuring filters to preprocess chunk data prior to compression. See the tutorial section on *Filters* and the *zarr.codecs* API docs for more information.

5.52.3 Other changes

To accommodate support for hierarchies and filters, the Zarr metadata format has been modified. See the *Zarr Storage Specification Version 2* for more information. To migrate an array stored using Zarr version 1.x, use the *zarr.storage.migrate_1to2()* function.

The bundled Blosc library has been upgraded to version 1.11.0.

5.52.4 Acknowledgments

Thanks to Matthew Rocklin, Stephan Hoyer and Francesc Alted for contributions and comments.

5.53 1.1.0

- The bundled Blosc library has been upgraded to version 1.10.0. The 'zstd' internal compression library is now available within Blosc. See the tutorial section on *Compressors* for an example.
- When using the Blosc compressor, the default internal compression library is now 'lz4'.
- The default number of internal threads for the Blosc compressor has been increased to a maximum of 8 (previously 4).
- Added convenience functions `zarr.blosc.list_compressors()` and `zarr.blosc.get_nthreads()`.

5.54 1.0.0

This release includes a complete re-organization of the code base. The major version number has been bumped to indicate that there have been backwards-incompatible changes to the API and the on-disk storage format. However, Zarr is still in an early stage of development, so please do not take the version number as an indicator of maturity.

5.54.1 Storage

The main motivation for re-organizing the code was to create an abstraction layer between the core array logic and data storage (#21). In this release, any object that implements the `MutableMapping` interface can be used as an array store. See the tutorial sections on *Persistent arrays* and *Storage alternatives*, the *Zarr Storage Specification Version 1*, and the `zarr.storage` module documentation for more information.

Please note also that the file organization and file name conventions used when storing a Zarr array in a directory on the file system have changed. Persistent Zarr arrays created using previous versions of the software will not be compatible with this version. See the `zarr.storage` API docs and the *Zarr Storage Specification Version 1* for more information.

5.54.2 Compression

An abstraction layer has also been created between the core array logic and the code for compressing and decompressing array chunks. This release still bundles the c-blosc library and uses Blosc as the default compressor, however other compressors including zlib, BZ2 and LZMA are also now supported via the Python standard library. New compressors can also be dynamically registered for use with Zarr. See the tutorial sections on *Compressors* and *Configuring Blosc*, the *Zarr Storage Specification Version 1*, and the `zarr.compressors` module documentation for more information.

5.54.3 Synchronization

The synchronization code has also been refactored to create a layer of abstraction, enabling Zarr arrays to be used in parallel computations with a number of alternative synchronization methods. For more information see the tutorial section on *Parallel computing and synchronization* and the `zarr.sync` module documentation.

5.54.4 Changes to the Blosc extension

NumPy is no longer a build dependency for the `zarr.blosc` Cython extension, so setup.py will run even if NumPy is not already installed, and should automatically install NumPy as a runtime dependency. Manual installation of NumPy prior to installing Zarr is still recommended, however, as the automatic installation of NumPy may fail or be sub-optimal on some platforms.

Some optimizations have been made within the `zarr.blosc` extension to avoid unnecessary memory copies, giving a ~10-20% performance improvement for multi-threaded compression operations.

The `zarr.blosc` extension now automatically detects whether it is running within a single-threaded or multi-threaded program and adapts its internal behaviour accordingly (#27). There is no need for the user to make any API calls to switch Blosc between contextual and non-contextual (global lock) mode. See also the tutorial section on *Configuring Blosc*.

5.54.5 Other changes

The internal code for managing chunks has been rewritten to be more efficient. Now no state is maintained for chunks outside of the array store, meaning that chunks do not carry any extra memory overhead not accounted for by the store. This negates the need for the "lazy" option present in the previous release, and this has been removed.

The memory layout within chunks can now be set as either "C" (row-major) or "F" (column-major), which can help to provide better compression for some data (#7). See the tutorial section on *Chunk memory layout* for more information.

A bug has been fixed within the `__getitem__` and `__setitem__` machinery for slicing arrays, to properly handle getting and setting partial slices.

5.54.6 Acknowledgments

Thanks to Matthew Rocklin, Stephan Hoyer, Francesc Alted, Anthony Scopatz and Martin Durant for contributions and comments.

5.55 0.4.0

See v0.4.0 release notes on GitHub.

5.56 0.3.0

See v0.3.0 release notes on GitHub.

CHAPTER

SIX

LICENSE

CHAPTER

SEVEN

ACKNOWLEDGMENTS

The following people have contributed to the development of Zarr by contributing code, documentation, code reviews, comments and/or ideas:

- Alistair Miles
- Altay Sansal
- Anderson Banihirwe
- Andrew Fulton
- Andrew Thomas
- Anthony Scopatz
- Attila Bergou
- BGCMHou
- Ben Jeffery
- Ben Williams
- Boaz Mohar
- Charles Noyes
- Chris Barnes
- David Baddeley
- Davis Bennett
- Dimitri Papadopoulos Orfanos
- Eduardo Gonzalez
- Elliott Sales de Andrade
- Eric Prestat
- Eric Younkin
- Francesc Alted
- Greggory Lee
- Gregory R. Lee
- Ian Hunt-Isaak
- James Bourbeau
- Jan Funke

- Jerome Kelleher
- Joe Hamman
- Joe Jevnik
- John Kirkham
- Josh Moore
- Juan Nunez-Iglesias
- Justin Swaney
- Mads R. B. Kristensen
- Mamy Ratsimbazafy
- Martin Durant
- Matthew Rocklin
- Matthias Bussonnier
- Mattia Almansi
- Noah D Brenowitz
- Oren Watson
- Pavithra Eswaramoorthy
- Poruri Sai Rahul
- Prakhar Goel
- Raphael Dussin
- Ray Bell
- Richard Scott
- Richard Shaw
- Ryan Abernathey
- Ryan Williams
- Saransh Chopra
- Sebastian Grill
- Shikhar Goenka
- Shivank Chaudhary
- Stephan Hoyer
- Stephan Saalfeld
- Tarik Onalan
- Tim Crone
- Tobias Kölling
- Tom Augspurger
- Tom White
- Tommy Tran

- Trevor Manz
- Vincent Schut
- Vyas Ramasubramani
- Zain Patel
- @gsakkis
- hailiangzhang
- pmav99
- sbalmer

CHAPTER

EIGHT

CONTRIBUTING TO ZARR

Zarr is a community maintained project. We welcome contributions in the form of bug reports, bug fixes, documentation, enhancement proposals and more. This page provides information on how best to contribute.

8.1 Asking for help

If you have a question about how to use Zarr, please post your question on StackOverflow using the "zarr" tag. If you don't get a response within a day or two, feel free to raise a GitHub issue including a link to your StackOverflow question. We will try to respond to questions as quickly as possible, but please bear in mind that there may be periods where we have limited time to answer questions due to other commitments.

8.2 Bug reports

If you find a bug, please raise a GitHub issue. Please include the following items in a bug report:

1. A minimal, self-contained snippet of Python code reproducing the problem. You can format the code nicely using markdown, e.g.:

```
```python
import zarr
g = zarr.group()
# etc.
```
```

2. An explanation of why the current behaviour is wrong/not desired, and what you expect instead.

3. Information about the version of Zarr, along with versions of dependencies and the Python interpreter, and installation information. The version of Zarr can be obtained from the `zarr.__version__` property. Please also state how Zarr was installed, e.g., "installed via pip into a virtual environment", or "installed using conda". Information about other packages installed can be obtained by executing `pip freeze` (if using pip to install packages) or `conda env export` (if using conda to install packages) from the operating system command prompt. The version of the Python interpreter can be obtained by running a Python interactive session, e.g.:

```
$ python
Python 3.6.1 (default, Mar 22 2017, 06:17:05)
[GCC 6.3.0 20170321] on linux
```

8.3 Enhancement proposals

If you have an idea about a new feature or some other improvement to Zarr, please raise a GitHub issue first to discuss.

We very much welcome ideas and suggestions for how to improve Zarr, but please bear in mind that we are likely to be conservative in accepting proposals for new features. The reasons for this are that we would like to keep the Zarr code base lean and focused on a core set of functionalities, and available time for development, review and maintenance of new features is limited. But if you have a great idea, please don't let that stop you from posting it on GitHub, just please don't be offended if we respond cautiously.

8.4 Contributing code and/or documentation

8.4.1 Forking the repository

The Zarr source code is hosted on GitHub at the following location:

- https://github.com/zarr-developers/zarr-python

You will need your own fork to work on the code. Go to the link above and hit the "Fork" button. Then clone your fork to your local machine:

```
$ git clone git@github.com:your-user-name/zarr-python.git
$ cd zarr-python
$ git remote add upstream git@github.com:zarr-developers/zarr-python.git
```

8.4.2 Creating a development environment

To work with the Zarr source code, it is recommended to set up a Python virtual environment and install all Zarr dependencies using the same versions as are used by the core developers and continuous integration services. Assuming you have a Python 3 interpreter already installed, and you have cloned the Zarr source code and your current working directory is the root of the repository, you can do something like the following:

```
$ mkdir -p ~/pyenv/zarr-dev
$ python -m venv ~/pyenv/zarr-dev
$ source ~/pyenv/zarr-dev/bin/activate
$ pip install -r requirements_dev_minimal.txt -r requirements_dev_numpy.txt
$ pip install -e .[docs]
```

To verify that your development environment is working, you can run the unit tests:

```
$ python -m pytest -v zarr
```

8.4.3 Creating a branch

Before you do any new work or submit a pull request, please open an issue on GitHub to report the bug or propose the feature you'd like to add.

It's best to synchronize your fork with the upstream repository, then create a new, separate branch for each piece of work you want to do. E.g.:

```
git checkout main
git fetch upstream
git rebase upstream/main
git push
git checkout -b shiny-new-feature
git push -u origin shiny-new-feature
```

This changes your working directory to the 'shiny-new-feature' branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to Zarr.

To update this branch with latest code from Zarr, you can retrieve the changes from the main branch and perform a rebase:

```
git fetch upstream
git rebase upstream/main
```

This will replay your commits on top of the latest Zarr git main. If this leads to merge conflicts, these need to be resolved before submitting a pull request. Alternatively, you can merge the changes in from upstream/main instead of rebasing, which can be simpler:

```
git fetch upstream
git merge upstream/main
```

Again, any conflicts need to be resolved before submitting a pull request.

8.4.4 Running the test suite

Zarr includes a suite of unit tests, as well as doctests included in function and class docstrings and in the tutorial and storage spec. The simplest way to run the unit tests is to activate your development environment (see *creating a development environment* above) and invoke:

```
$ python -m pytest -v zarr
```

Some tests require optional dependencies to be installed, otherwise the tests will be skipped. To install all optional dependencies, run:

```
$ pip install -r requirements_dev_optional.txt
```

To also run the doctests within docstrings (requires optional dependencies to be installed), run:

```
$ python -m pytest -v --doctest-plus zarr
```

To run the doctests within the tutorial and storage spec (requires optional dependencies to be installed), run:

```
$ python -m doctest -o NORMALIZE_WHITESPACE -o ELLIPSIS docs/tutorial.rst docs/spec/v2.
↪rst
```

Note that some tests also require storage services to be running locally. To run the Azure Blob Service storage tests, run an Azure storage emulator (e.g., azurite) and set the environment variable `ZARR_TEST_ABS=1`. If you're using Docker to run azurite, start the service with:

```
docker run --rm -p 10000:10000 mcr.microsoft.com/azure-storage/azurite azurite-blob --
↪loose --blobHost 0.0.0.0
```

To run the Mongo DB storage tests, run a Mongo server locally and set the environment variable `ZARR_TEST_MONGO=1`. To run the Redis storage tests, run a Redis server locally on port 6379 and set the environment variable `ZARR_TEST_REDIS=1`.

All tests are automatically run via GitHub Actions for every pull request and must pass before code can be accepted. Test coverage is also collected automatically via the Codecov service, and total coverage over all builds must be 100% (although individual builds may be lower due to Python 2/3 or other differences).

8.4.5 Code standards - using pre-commit

All code must conform to the PEP8 standard. Regarding line length, lines up to 100 characters are allowed, although please try to keep under 90 wherever possible.

`Zarr` uses a set of `pre-commit` hooks and the `pre-commit` bot to format, type-check, and prettify the codebase. `pre-commit` can be installed locally by running:

```
$ python -m pip install pre-commit
```

The hooks can be installed locally by running:

```
$ pre-commit install
```

This would run the checks every time a commit is created locally. These checks will also run on every commit pushed to an open PR, resulting in some automatic styling fixes by the `pre-commit` bot. The checks will by default only run on the files modified by a commit, but the checks can be triggered for all the files by running:

```
$ pre-commit run --all-files
```

If you would like to skip the failing checks and push the code for further discussion, use the `--no-verify` option with `git commit`.

8.4.6 Test coverage

Zarr maintains 100% test coverage under the latest Python stable release (currently Python 3.8). Both unit tests and docstring doctests are included when computing coverage. Running:

```
$ python -m pytest -v --cov=zarr --cov-config=pyproject.toml zarr
```

will automatically run the test suite with coverage and produce a coverage report. This should be 100% before code can be accepted into the main code base.

When submitting a pull request, coverage will also be collected across all supported Python versions via the Codecov service, and will be reported back within the pull request. Codecov coverage must also be 100% before code can be accepted.

8.4.7 Documentation

Docstrings for user-facing classes and functions should follow the numpydoc standard, including sections for Parameters and Examples. All examples should run and pass as doctests under Python 3.8. To run doctests, activate your development environment, install optional requirements, and run:

```
$ python -m pytest -v --doctest-plus zarr
```

Zarr uses Sphinx for documentation, hosted on readthedocs.org. Documentation is written in the RestructuredText markup language (.rst files) in the `docs` folder. The documentation consists both of prose and API documentation. All user-facing classes and functions should be included in the API documentation, under the `docs/api` folder. Any new features or important usage information should be included in the tutorial (`docs/tutorial.rst`). Any changes should also be included in the release notes (`docs/release.rst`).

The documentation can be built locally by running:

```
$ cd docs
$ make clean; make html
$ open _build/html/index.html
```

The resulting built documentation will be available in the `docs/_build/html` folder.

8.5 Development best practices, policies and procedures

The following information is mainly for core developers, but may also be of interest to contributors.

8.5.1 Merging pull requests

Pull requests submitted by an external contributor should be reviewed and approved by at least one core developers before being merged. Ideally, pull requests submitted by a core developer should be reviewed and approved by at least one other core developers before being merged.

Pull requests should not be merged until all CI checks have passed (GitHub Actions Codecov) against code that has had the latest main merged in.

8.5.2 Compatibility and versioning policies

Because Zarr is a data storage library, there are two types of compatibility to consider: API compatibility and data format compatibility.

API compatibility

All functions, classes and methods that are included in the API documentation (files under `docs/api/*.rst`) are considered as part of the Zarr **public API**, except if they have been documented as an experimental feature, in which case they are part of the **experimental API**.

Any change to the public API that does **not** break existing third party code importing Zarr, or cause third party code to behave in a different way, is a **backwards-compatible API change**. For example, adding a new function, class or method is usually a backwards-compatible change. However, removing a function, class or method; removing an argument to a function or method; adding a required argument to a function or method; or changing the behaviour of a function or method, are examples of **backwards-incompatible API changes**.

If a release contains no changes to the public API (e.g., contains only bug fixes or other maintenance work), then the micro version number should be incremented (e.g., 2.2.0 -> 2.2.1). If a release contains public API changes, but all changes are backwards-compatible, then the minor version number should be incremented (e.g., 2.2.1 -> 2.3.0). If a release contains any backwards-incompatible public API changes, the major version number should be incremented (e.g., 2.3.0 -> 3.0.0).

Backwards-incompatible changes to the experimental API can be included in a minor release, although this should be minimised if possible. I.e., it would be preferable to save up backwards-incompatible changes to the experimental API to be included in a major release, and to stabilise those features at the same time (i.e., move from experimental to public API), rather than frequently tinkering with the experimental API in minor releases.

Data format compatibility

The data format used by Zarr is defined by a specification document, which should be platform-independent and contain sufficient detail to construct an interoperable software library to read and/or write Zarr data using any programming language. The latest version of the specification document is available from the *Specifications* page.

Here, **data format compatibility** means that all software libraries that implement a particular version of the Zarr storage specification are interoperable, in the sense that data written by any one library can be read by all others. It is obviously desirable to maintain data format compatibility wherever possible. However, if a change is needed to the storage specification, and that change would break data format compatibility in any way, then the storage specification version number should be incremented (e.g., 2 -> 3).

The versioning of the Zarr software library is related to the versioning of the storage specification as follows. A particular version of the Zarr library will implement a particular version of the storage specification. For example, Zarr version 2.2.0 implements the Zarr storage specification version 2. If a release of the Zarr library implements a different version of the storage specification, then the major version number of the Zarr library should be incremented. E.g., if Zarr version 2.2.0 implements the storage spec version 2, and the next release of the Zarr library implements storage spec version 3, then the next library release should have version number 3.0.0. Note however that the major version number of the Zarr library may not always correspond to the spec version number. For example, Zarr versions 2.x, 3.x, and 4.x might all implement the same version of the storage spec and thus maintain data format compatibility, although they will not maintain API compatibility. The version number of the storage specification that is currently implemented is stored under the `zarr.meta.ZARR_FORMAT` variable.

Note that the Zarr test suite includes a data fixture and tests to try and ensure that data format compatibility is not accidentally broken. See the `test_format_compatibility()` function in the `zarr.tests.test_storage` module for details.

8.5.3 When to make a release

Ideally, any bug fixes that don't change the public API should be released as soon as possible. It is fine for a micro release to contain only a single bug fix.

When to make a minor release is at the discretion of the core developers. There are no hard-and-fast rules, e.g., it is fine to make a minor release to make a single new feature available; equally, it is fine to make a minor release that includes a number of changes.

Major releases obviously need to be given careful consideration, and should be done as infrequently as possible, as they will break existing code and/or affect data compatibility in some way.

8.5.4 Release procedure

Note: Most of the release process is now handled by github workflow which should automatically push a release to PyPI if a tag is pushed.

Before releasing, make sure that all pull requests which will be included in the release have been properly documented in *docs/release.rst*.

To make a new release, go to https://github.com/zarr-developers/zarr-python/releases and click "Draft a new release". Choose a version number prefixed with a *v* (e.g. *v0.0.0*). For pre-releases, include the appropriate suffix (e.g. *v0.0.0a1* or *v0.0.0rc2*).

Set the description of the release to:

```
See release notes https://zarr.readthedocs.io/en/stable/release.html#release-0-0-0
```

replacing the correct version numbers. For pre-release versions, the URL should omit the pre-release suffix, e.g. "a1" or "rc1".

Click on "Generate release notes" to auto-file the description.

After creating the release, the documentation will be built on https://readthedocs.io. Full releases will be available under /stable while pre-releases will be available under /latest.

Also review and merge the https://github.com/conda-forge/zarr-feedstock pull request that will be automatically generated.

Version: 2.17.1

Download documentation: PDF/Zipped HTML

Useful links: Installation | Source Repository | Issue Tracker | Zulip Chat

Zarr is a file storage format for chunked, compressed, N-dimensional arrays based on an open-source specification.

Getting Started

New to Zarr? Check out the getting started guide. It contains an introduction to Zarr's main concepts and links to additional tutorials.

To the getting started guide

Tutorial

The tutorial provides working examples of Zarr classes and functions.

To the Tutorial

API Reference

The reference guide contains a detailed description of the functions, modules, and objects included in Zarr. The reference describes how the methods work and which parameters can be used. It assumes that you have an understanding of the key concepts.

To the api reference guide

Contributor's Guide

Want to contribute to Zarr? We welcome contributions in the form of bug reports, bug fixes, documentation, enhancement proposals and more. The contributing guidelines will guide you through the process of improving Zarr.

To the contributor's guide

PYTHON MODULE INDEX

Z

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z