h5py Documentation

Release 3.10.0

Andrew Collette and contributors

Mar 24, 2024

CONTENTS

1 Where to start 3

2 Other resources 5

3 Introductory info 7

4 High-level API reference 15

5 Advanced topics 47

6 Meta-info about the h5py project 69

Index 113

The h5py package is a Pythonic interface to the HDF5 binary data format.

HDF5 lets you store huge amounts of numerical data, and easily manipulate that data from NumPy. For example, you can slice into multi-terabyte datasets stored on disk, as if they were real NumPy arrays. Thousands of datasets can be stored in a single file, categorized and tagged however you want.

CHAPTER

ONE

WHERE TO START

- *Quick-start guide*
- *Installation*

CHAPTER

TWO

OTHER RESOURCES

- Python and HDF5 O'Reilly book
- Ask questions on the HDF forum
- GitHub project

CHAPTER

THREE

INTRODUCTORY INFO

3.1 Quick Start Guide

3.1.1 Install

With Anaconda or Miniconda:

```
conda install h5py
```

If there are wheels for your platform (mac, linux, windows on x86) and you do not need MPI you can install `h5py` via pip:

```
pip install h5py
```

With Enthought Canopy, use the GUI package manager or:

```
enpkg h5py
```

To install from source see *Installation*.

3.1.2 Core concepts

An HDF5 file is a container for two kinds of objects: *datasets*, which are array-like collections of data, and *groups*, which are folder-like containers that hold datasets and other groups. The most fundamental thing to remember when using h5py is:

> **Groups work like dictionaries, and datasets work like NumPy arrays**

Suppose someone has sent you a HDF5 file, `mytestfile.hdf5`. (To create this file, read *Appendix: Creating a file*.) The very first thing you'll need to do is to open the file for reading:

```
>>> import h5py
>>> f = h5py.File('mytestfile.hdf5', 'r')
```

The *File object* is your starting point. What is stored in this file? Remember *h5py.File* acts like a Python dictionary, thus we can check the keys,

```
>>> list(f.keys())
['mydataset']
```

Based on our observation, there is one data set, `mydataset` in the file. Let us examine the data set as a *Dataset* object

```
>>> dset = f['mydataset']
```

The object we obtained isn't an array, but *an HDF5 dataset*. Like NumPy arrays, datasets have both a shape and a data type:

```
>>> dset.shape
(100,)
>>> dset.dtype
dtype('int32')
```

They also support array-style slicing. This is how you read and write data from a dataset in the file:

```
>>> dset[...] = np.arange(100)
>>> dset[0]
0
>>> dset[10]
10
>>> dset[0:100:10]
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

For more, see *File Objects* and *Datasets*.

Appendix: Creating a file

At this point, you may wonder how `mytestdata.hdf5` is created. We can create a file by setting the `mode` to `w` when the File object is initialized. Some other modes are `a` (for read/write/create access), and `r+` (for read/write access). A full list of file access modes and their meanings is at *File Objects*.

```
>>> import h5py
>>> import numpy as np
>>> f = h5py.File("mytestfile.hdf5", "w")
```

The *File object* has a couple of methods which look interesting. One of them is `create_dataset`, which as the name suggests, creates a data set of given shape and dtype

```
>>> dset = f.create_dataset("mydataset", (100,), dtype='i')
```

The File object is a context manager; so the following code works too

```
>>> import h5py
>>> import numpy as np
>>> with h5py.File("mytestfile.hdf5", "w") as f:
>>>     dset = f.create_dataset("mydataset", (100,), dtype='i')
```

3.1.3 Groups and hierarchical organization

"HDF" stands for "Hierarchical Data Format". Every object in an HDF5 file has a name, and they're arranged in a POSIX-style hierarchy with /-separators:

```
>>> dset.name
'/mydataset'
```

The "folders" in this system are called *groups*. The `File` object we created is itself a group, in this case the *root group*, named /:

```
>>> f.name
'/'
```

Creating a subgroup is accomplished via the aptly-named `create_group`. But we need to open the file in the "append" mode first (Read/write if exists, create otherwise)

```
>>> f = h5py.File('mydataset.hdf5', 'a')
>>> grp = f.create_group("subgroup")
```

All `Group` objects also have the `create_*` methods like File:

```
>>> dset2 = grp.create_dataset("another_dataset", (50,), dtype='f')
>>> dset2.name
'/subgroup/another_dataset'
```

By the way, you don't have to create all the intermediate groups manually. Specifying a full path works just fine:

```
>>> dset3 = f.create_dataset('subgroup2/dataset_three', (10,), dtype='i')
>>> dset3.name
'/subgroup2/dataset_three'
```

Groups support most of the Python dictionary-style interface. You retrieve objects in the file using the item-retrieval syntax:

```
>>> dataset_three = f['subgroup2/dataset_three']
```

Iterating over a group provides the names of its members:

```
>>> for name in f:
...     print(name)
mydataset
subgroup
subgroup2
```

Membership testing also uses names:

```
>>> "mydataset" in f
True
>>> "somethingelse" in f
False
```

You can even use full path names:

```
>>> "subgroup/another_dataset" in f
True
```

There are also the familiar `keys()`, `values()`, `items()` and `iter()` methods, as well as `get()`.

Since iterating over a group only yields its directly-attached members, iterating over an entire file is accomplished with the `Group` methods `visit()` and `visititems()`, which take a callable:

```
>>> def printname(name):
...     print(name)
>>> f.visit(printname)
mydataset
subgroup
subgroup/another_dataset
subgroup2
subgroup2/dataset_three
```

For more, see *Groups*.

3.1.4 Attributes

One of the best features of HDF5 is that you can store metadata right next to the data it describes. All groups and datasets support attached named bits of data called *attributes*.

Attributes are accessed through the `attrs` proxy object, which again implements the dictionary interface:

```
>>> dset.attrs['temperature'] = 99.5
>>> dset.attrs['temperature']
99.5
>>> 'temperature' in dset.attrs
True
```

For more, see *Attributes*.

3.2 Installation

It is highly recommended that you use a pre-built version of h5py, either from a Python Distribution, an OS-specific package manager, or a pre-built wheel from PyPI.

Be aware however that most pre-built versions lack MPI support, and that they are built against a specific version of HDF5. If you require MPI support, or newer HDF5 features, you will need to build from source.

After installing h5py, you should run the tests to be sure that everything was installed correctly. This can be done in the python interpreter via:

```
import h5py
h5py.run_tests()
```

3.2.1 Pre-built installation (recommended)

Pre-build h5py can be installed via many Python Distributions, OS-specific package managers, or via h5py wheels.

Python Distributions

If you do not already use a Python Distribution, we recommend either Anaconda/Miniconda or Enthought Canopy, both of which support most versions of Microsoft Windows, OSX/MacOS, and a variety of Linux Distributions. Installation of h5py can be done on the command line via:

```
$ conda install h5py
```

for Anaconda/MiniConda, and via:

```
$ enpkg h5py
```

for Canopy.

Wheels

If you have an existing Python installation (e.g. a python.org download, or one that comes with your OS), then on Windows, MacOS/OSX, and Linux on Intel computers, pre-built h5py wheels can be installed via pip from PyPI:

```
$ pip install h5py
```

Additionally, for Windows users, Chris Gohlke provides third-party wheels which use Intel's MKL.

OS-Specific Package Managers

On OSX/MacOS, h5py can be installed via Homebrew, Macports, or Fink.

The current state of h5py in various Linux Distributions can be seen at https://pkgs.org/download/python-h5py, and can be installed via the package manager.

As far as the h5py developers know, none of the Windows package managers (e.g. Chocolatey, nuget) have h5py included, however they may assist in installing h5py's requirements when building from source.

3.2.2 Source installation

To install h5py from source, you need:

- A supported Python version with development headers
- HDF5 1.10.4 or newer with development headers
  - HDF5 versions newer than the h5py version you're using might not work.
  - Odd minor versions of HDF5 (e.g. 1.13) are experimental, and might not work. Use a 'maintenance' version like 1.12.x if possible.
  - If you need support for older HDF5 versions, h5py up to version 3.9 supported HDF5 1.8.4 and above.
- A C compiler

On Unix platforms, you also need `pkg-config` unless you explicitly specify a path for HDF5 as described in *Custom installation*.

There are notes below on installing HDF5, Python and a C compiler on different platforms.

Building h5py also requires several Python packages, but in most cases pip will automatically install these in a build environment for you, so you don't need to deal with them manually. See *Development installation* for a list.

The actual installation of h5py should be done via:

```
$ pip install --no-binary=h5py h5py
```

or, from a tarball or git *checkout*:

```
$ pip install -v .
```

Development installation

When modifying h5py, you often want to reinstall it quickly to test your changes. To benefit from caching and use NumPy & Cython from your existing Python environment, run:

```
$ H5PY_SETUP_REQUIRES=0 python3 setup.py build
$ python3 -m pip install . --no-build-isolation
```

For convenience, these commands are also in a script `dev-install.sh` in the h5py git repository.

This skips setting up a build environment, so you should have already installed Cython, NumPy, pkgconfig (a Python interface to `pkg-config`) and mpi4py (if you want MPI integration - see *Building against Parallel HDF5*). See `setup.py` for minimum versions.

This will normally rebuild Cython files automatically when they change, but sometimes it may be necessary to force a full rebuild. The easiest way to achieve this is to discard everything but the code committed to git. In the root of your git checkout, run:

```
$ git clean -xfd
```

Then build h5py again as above.

Source installation on OSX/MacOS

HDF5 and Python are most likely in your package manager (e.g. Homebrew, Macports, or Fink). Be sure to install the development headers, as sometimes they are not included in the main package.

XCode comes with a C compiler (clang), and your package manager will likely have other C compilers for you to install.

Source installation on Linux/Other Unix

HDF5 and Python are most likely in your package manager. A C compiler almost definitely is, usually there is some kind of metapackage to install the default build tools, e.g. `build-essential`, which should be sufficient for our needs. Make sure that that you have the development headers, as they are usually not installed by default. They can usually be found in `python-dev` or similar and `libhdf5-dev` or similar.

Source installation on Windows

Installing from source on Windows is a much more difficult prospect than installing from source on other OSs, as not only are you likely to need to compile HDF5 from source, everything must be built with the correct version of Visual Studio. Additional patches are also needed to HDF5 to get HDF5 and Python to work together.

We recommend examining the appveyor build scripts, and using those to build and install HDF5 and h5py.

Downstream packagers

If you are building h5py for another packaging system - e.g. Linux distros or packaging aimed at HPC users - you probably want to satisfy build dependencies from your packaging system. To build without automatically fetching dependencies, use a command like:

```
H5PY_SETUP_REQUIRES=0 pip install . --no-deps --no-build-isolation
```

Depending on your packaging system, you may need to use the `--prefix` or `--root` options to control where files get installed.

h5py's Python packaging has build dependencies on the oldest compatible versions of NumPy and mpi4py. You can build with newer versions of these, but the resulting h5py binaries will only work with the NumPy & mpi4py versions they were built with (or newer). Mpi4py is an optional dependency, only required for *Parallel HDF5* features.

You should also look at the build options under *Custom installation*.

3.2.3 Custom installation

Important: Remember that pip installs wheels by default. To perform a custom installation with pip, you should use:

```
$ pip install --no-binary=h5py h5py
```

or build from a git checkout or downloaded tarball to avoid getting a pre-built version of h5py.

You can specify build options for h5py as environment variables when you build it from source:

```
$ HDF5_DIR=/path/to/hdf5 pip install --no-binary=h5py h5py
$ HDF5_VERSION=X.Y.Z pip install --no-binary=h5py h5py
$ CC="mpicc" HDF5_MPI="ON" HDF5_DIR=/path/to/parallel-hdf5 pip install --no-binary=h5py␣
↪h5py
```

The supported build options are:

- To specify where to find HDF5, use one of these options:
  - `HDF5_LIBDIR` and `HDF5_INCLUDEDIR`: the directory containing the compiled HDF5 libraries and the directory containing the C header files, respectively.
  - `HDF5_DIR`: a shortcut for common installations, a directory with `lib` and `include` subdirectories containing compiled libraries and C headers.
  - `HDF5_PKGCONFIG_NAME`: A name to query `pkg-config` for. If none of these options are specified, h5py will query `pkg-config` by default for `hdf5`, or `hdf5-openmpi` if building with MPI support.
- `HDF5_MPI=ON` to build with MPI integration - see *Building against Parallel HDF5*.

- `HDF5_VERSION` to force a specified HDF5 version. In most cases, you don't need to set this; the version number will be detected from the HDF5 library.
- `H5PY_SYSTEM_LZF=1` to build the bundled LZF compression filter (see *Filter pipeline*) against an external LZF library, rather than using the bundled LZF C code.

3.2.4 Building against Parallel HDF5

If you just want to build with `mpicc`, and don't care about using Parallel HDF5 features in h5py itself:

```
$ export CC=mpicc
$ pip install --no-binary=h5py h5py
```

If you want access to the full Parallel HDF5 feature set in h5py (*Parallel HDF5*), you will further have to build in MPI mode. This can be done by setting the `HDF5_MPI` environment variable:

```
$ export CC=mpicc
$ export HDF5_MPI="ON"
$ pip install --no-binary=h5py h5py
```

You will need a shared-library build of Parallel HDF5 as well, i.e. built with `./configure --enable-shared --enable-parallel`.

On Windows, MS-MPI is usually used which does not have an `mpicc` wrapper. Instead, you may use the `H5PY_MSMPI` environment variable to `ON` in order to query the system for MS-MPI's information.

CHAPTER

FOUR

HIGH-LEVEL API REFERENCE

4.1 File Objects

File objects serve as your entry point into the world of HDF5. In addition to the File-specific capabilities listed here, every File instance is also an *HDF5 group* representing the *root group* of the file.

4.1.1 Opening & creating files

HDF5 files work generally like standard Python file objects. They support standard modes like r/w/a, and should be closed when they are no longer in use. However, there is obviously no concept of "text" vs "binary" mode.

```
>>> f = h5py.File('myfile.hdf5','r')
```

The file name may be a byte string or unicode string. Valid modes are:

r	Readonly, file must exist (default)
r+	Read/write, file must exist
w	Create file, truncate if exists
w- or x	Create file, fail if exists
a	Read/write if exists, create otherwise

Changed in version 3.0: Files are now opened read-only by default. Earlier versions of h5py would pick different modes depending on the presence and permissions of the file.

4.1.2 File drivers

HDF5 ships with a variety of different low-level drivers, which map the logical HDF5 address space to different storage mechanisms. You can specify which driver you want to use when the file is opened:

```
>>> f = h5py.File('myfile.hdf5', driver=<driver name>, <driver_kwds>)
```

For example, the HDF5 "core" driver can be used to create a purely in-memory HDF5 file, optionally written out to disk when it is closed. Here's a list of supported drivers and their options:

None
: **Strongly recommended.** Use the standard HDF5 driver appropriate for the current platform. On UNIX, this is the H5FD_SEC2 driver; on Windows, it is H5FD_WINDOWS.

'sec2'
: Unbuffered, optimized I/O using standard POSIX functions.

'stdio'
Buffered I/O using functions from stdio.h.

'core'
Store and manipulate the data in memory, and optionally write it back out when the file is closed. Using this with an existing file and a reading mode will read the entire file into memory. Keywords:

backing_store:
If True (default), save changes to the real file at the specified path on *close()* or *flush()*. If False, any changes are discarded when the file is closed.

block_size:
Increment (in bytes) by which memory is extended. Default is 64k.

'family'
Store the file on disk as a series of fixed-length chunks. Useful if the file system doesn't allow large files. Note: the filename you provide *must* contain a printf-style integer format code (e.g. %d"), which will be replaced by the file sequence number. Keywords:

memb_size: Maximum file size (default is 2**31-1).

'fileobj'
Store the data in a Python file-like object; see below. This is the default if a file-like object is passed to *File*.

'split'
Splits the meta data and raw data into separate files. Keywords:

meta_ext:
Metadata filename extension. Default is '-m.h5'.

raw_ext:
Raw data filename extension. Default is '-r.h5'.

'ros3'
Enables read-only access to HDF5 files in the AWS S3 or S3-compatible object stores. HDF5 file name must be one of http://, https://, or s3:// resource location. An s3:// location will be translated into an AWS path-style location by h5py. Keywords:

aws_region:
AWS region of the S3 bucket with the file, e.g. `b"us-east-1"`. Default is `b''`. Required for s3:// locations.

secret_id:
AWS access key ID. Default is `b''`.

secret_key:
AWS secret access key. Default is `b''`.

session_token:
AWS temporary session token. Default is `b''`.' Must be used together with temporary secret_id and secret_key. Available from HDF5 1.14.2.

The argument values must be `bytes` objects. Arguments aws_region, secret_id, and secret_key are required to activate AWS authentication.

Note: Pre-built h5py packages on PyPI do not include ros3 driver support. If you want this feature, you could use packages from conda-forge, or *build h5py from source* against an HDF5 build with ros3. Alternatively, use the *file-like object* support with a package like s3fs.

4.1.3 Python file-like objects

New in version 2.9.

The first argument to *File* may be a Python file-like object, such as an `io.BytesIO` or `tempfile.TemporaryFile` instance. This is a convenient way to create temporary HDF5 files, e.g. for testing or to send over the network.

The file-like object must be open for binary I/O, and must have these methods: `read()` (or `readinto()`), `write()`, `seek()`, `tell()`, `truncate()` and `flush()`.

```
>>> tf = tempfile.TemporaryFile()
>>> f = h5py.File(tf, 'w')
```

Accessing the *File* instance after the underlying file object has been closed will result in undefined behaviour.

When using an in-memory object such as `io.BytesIO`, the data written will take up space in memory. If you want to write large amounts of data, a better option may be to store temporary data on disk using the functions in `tempfile`.

```
"""Create an HDF5 file in memory and retrieve the raw bytes

This could be used, for instance, in a server producing small HDF5
files on demand.
"""
import io
import h5py

bio = io.BytesIO()
with h5py.File(bio, 'w') as f:
    f['dataset'] = range(10)

data = bio.getvalue() # data is a regular Python bytes object.
print("Total size:", len(data))
print("First bytes:", data[:10])
```

Warning: When using a Python file-like object for an HDF5 file, make sure to close the HDF5 file before closing the file object it's wrapping. If there is an error while trying to close the HDF5 file, segfaults may occur.

Warning: When using a Python file-like object, using service threads to implement the file-like API can lead to process deadlocks.

`h5py` serializes access to low-level hdf5 functions via a global lock. This lock is held when the file-like methods are called and is required to delete/deallocate `h5py` objects. Thus, if cyclic garbage collection is triggered on a service thread the program will deadlock. The service thread can not continue until it acquires the lock, and the thread holding the lock will not release it until the service thread completes its work.

If possible, avoid creating circular references (either via `weakrefs` or manually breaking the cycles) that keep `h5py` objects alive. If this is not possible, manually triggering a garbage collection from the correct thread or temporarily disabling garbage collection may help.

Note: Using a Python file-like object for HDF5 is internally more complex, as the HDF5 C code calls back into Python to access it. It inevitably has more ways to go wrong, and the failures may be less clear when it does. For some common use cases, you can easily avoid it:

- To create a file in memory and never write it to disk, use the `'core'` driver with `mode='w'`, `backing_store=False` (see *File drivers*).
- To use a temporary file securely, make a temporary directory and *open a file path* inside it.

4.1.4 Version bounding

HDF5 has been evolving for many years now. By default, the library will write objects in the most compatible fashion possible, so that older versions will still be able to read files generated by modern programs. However, there can be feature or performance advantages if you are willing to forgo a certain level of backwards compatibility. By using the "libver" option to *File*, you can specify the minimum and maximum sophistication of these structures:

```
>>> f = h5py.File('name.hdf5', libver='earliest') # most compatible
>>> f = h5py.File('name.hdf5', libver='latest')   # most modern
```

Here "latest" means that HDF5 will always use the newest version of these structures without particular concern for backwards compatibility. The "earliest" option means that HDF5 will make a *best effort* to be backwards compatible.

The default is "earliest".

Specifying version bounds has changed from HDF5 version 1.10.2. There are two new compatibility levels: *v108* (for HDF5 1.8) and *v110* (for HDF5 1.10). This change enables, for example, something like this:

```
>>> f = h5py.File('name.hdf5', libver=('earliest', 'v108'))
```

which enforces full backward compatibility up to HDF5 1.8. Using any HDF5 feature that requires a newer format will raise an error.

latest is now an alias to another bound label that represents the latest version. Because of this, the *File.libver* property will not use *latest* in its output for HDF5 1.10.2 or later.

4.1.5 Closing files

If you call *File.close()*, or leave a `with h5py.File(...)` block, the file will be closed and any objects (such as groups or datasets) you have from that file will become unusable. This is equivalent to what HDF5 calls 'strong' closing.

If a file object goes out of scope in your Python code, the file will only be closed when there are no remaining objects belonging to it. This is what HDF5 calls 'weak' closing.

```
with h5py.File('f1.h5', 'r') as f1:
    ds = f1['dataset']

# ERROR - can't access dataset, because f1 is closed:
ds[0]

def get_dataset():
    f2 = h5py.File('f2.h5', 'r')
    return f2['dataset']
ds = get_dataset()

# OK - f2 is out of scope, but the dataset reference keeps it open:
ds[0]
```

(continues on next page)

(continued from previous page)

```
del ds  # Now f2.h5 will be closed
```

4.1.6 User block

HDF5 allows the user to insert arbitrary data at the beginning of the file, in a reserved space called the *user block*. The length of the user block must be specified when the file is created. It can be either zero (the default) or a power of two greater than or equal to 512. You can specify the size of the user block when creating a new file, via the `userblock_size` keyword to File; the userblock size of an open file can likewise be queried through the `File.userblock_size` property.

Modifying the user block on an open file is not supported; this is a limitation of the HDF5 library. However, once the file is closed you are free to read and write data at the start of the file, provided your modifications don't leave the user block region.

4.1.7 Filenames on different systems

Different operating systems (and different file systems) store filenames with different encodings. Additionally, in Python there are at least two different representations of filenames, as encoded `bytes` or as a Unicode string (`str` on Python 3).

h5py's high-level interfaces always return filenames as `str`, e.g. *`File.filename`*. h5py accepts filenames as either `str` or `bytes`. In most cases, using Unicode (`str`) paths is preferred, but there are some caveats.

Note: HDF5 handles filenames as bytes (C `char *`), and the h5py *Low-Level API* matches this.

---

macOS (OSX)

macOS is the simplest system to deal with, it only accepts UTF-8, so using Unicode paths will just work (and should be preferred).

Linux (and non-macOS Unix)

Filenames on Unix-like systems are natively bytes. By convention, the locale encoding is used to convert to and from unicode; on most modern systems this will be UTF-8 by default (especially since Python 3.7, with **PEP 538**).

Passing Unicode paths will mostly work, and Unicode paths from system functions like `os.listdir()` should always work. But if there are filenames that aren't in the expected encoding (e.g. on a network filesystem or a removable drive, or because something is misconfigured), you may want to handle them as bytes.

Windows

Windows systems natively handle filenames as Unicode, and with HDF5 1.10.6 and above filenames passed to h5py as bytes will be used as UTF-8 encoded text, regardless of system configuration.

HDF5 1.10.5 and below could only use filenames with characters from the active code page, e.g. Windows-1252 on many systems configured for European languages. This limitation applies whether you use `str` or `bytes` with h5py.

4.1.8 Chunk cache

Chunked storage allows datasets to be stored on disk in separate pieces. When a part of any one of these pieces is needed, the entire chunk is read into memory before the requested part is copied to the user's buffer. To the extent possible those chunks are cached in memory, so that if the user requests a different part of a chunk that has already been read, the data can be copied directly from memory rather than reading the file again. The details of a given dataset's chunks are controlled when creating the dataset, but it is possible to adjust the behavior of the chunk *cache* when opening the file.

The parameters controlling this behavior are prefixed by `rdcc`, for *raw data chunk cache*. They apply to all datasets unless specifically changed for each one.

- `rdcc_nbytes` sets the total size (measured in bytes) of the raw data chunk cache for each dataset. The default size is 1 MiB. This should be set to the size of each chunk times the number of chunks that are likely to be needed in cache.
- `rdcc_w0` sets the policy for chunks to be removed from the cache when more space is needed. If the value is set to 0, then the library will always evict the least recently used chunk in cache. If the value is set to 1, the library will always evict the least recently used chunk which has been fully read or written, and if none have been fully read or written, it will evict the least recently used chunk. If the value is between 0 and 1, the behavior will be a blend of the two. Therefore, if the application will access the same data more than once, the value should be set closer to 0, and if the application does not, the value should be set closer to 1.
- `rdcc_nslots` is the number of chunk slots in the cache for each dataset. In order to allow the chunks to be looked up quickly in cache, each chunk is assigned a unique hash value that is used to look up the chunk. The cache contains a simple array of pointers to chunks, which is called a hash table. A chunk's hash value is simply the index into the hash table of the pointer to that chunk. While the pointer at this location might instead point to a different chunk or to nothing at all, no other locations in the hash table can contain a pointer to the chunk in question. Therefore, the library only has to check this one location in the hash table to tell if a chunk is in cache or not. This also means that if two or more chunks share the same hash value, then only one of those chunks can be in the cache at the same time. When a chunk is brought into cache and another chunk with the same hash value is already in cache, the second chunk must be evicted first. Therefore it is very important to make sure that the size of the hash table (which is determined by the `rdcc_nslots` parameter) is large enough to minimize the number of hash value collisions. Due to the hashing strategy, this value should ideally be a prime number. As a rule of thumb, this value should be at least 10 times the number of chunks that can fit in `rdcc_nbytes` bytes. For maximum performance, this value should be set approximately 100 times that number of chunks. The default value is 521.

Chunks and caching are described in greater detail in the HDF5 documentation.

4.1.9 Data alignment

When creating datasets within files, it may be advantageous to align the offset within the file itself. This can help optimize read and write times if the data become aligned with the underlying hardware, or may help with parallelism with MPI. Unfortunately, aligning small variables to large blocks can leave a lot of empty space in a file. To this effect, application developers are left with two options to tune the alignment of data within their file. The two variables `alignment_threshold` and `alignment_interval` in the *File* constructor help control the threshold in bytes where the data alignment policy takes effect and the alignment in bytes within the file. The alignment is measured from the end of the user block.

For more information, see the official HDF5 documentation H5P_SET_ALIGNMENT.

4.1.10 Meta block size

Space for metadata is allocated in blocks within the HDF5 file. The argument `meta_block_size` of the *File* constructor sets the minimum size of these blocks. Setting a large value can consolidate metadata into a small number of regions. Setting a small value can reduce the overall file size, especially in combination with the `libver` option. This controls how the overall data and metadata are laid out within the file.

For more information, see the official HDF5 documentation H5P_SET_META_BLOCK_SIZE.

4.1.11 Reference

Note: Unlike Python file objects, the attribute `File.name` gives the HDF5 name of the root group, "/". To access the on-disk name, use *File.filename*.

---

class h5py.**File**(*name*, *mode='r'*, *driver=None*, *libver=None*, *userblock_size=None*, *swmr=False*, *rdcc_nslots=None*, *rdcc_nbytes=None*, *rdcc_w0=None*, *track_order=None*, *fs_strategy=None*, *fs_persist=False*, *fs_threshold=1*, *fs_page_size=None*, *page_buf_size=None*, *min_meta_keep=0*, *min_raw_keep=0*, *locking=None*, *alignment_threshold=1*, *alignment_interval=1*, ***kwds*)

Open or create a new file.

Note that in addition to the *File*-specific methods and properties listed below, *File* objects inherit the full interface of *Group*.

Parameters

- **name** – Name of file (*bytes* or *str*), or an instance of `h5f.FileID` to bind to an existing file identifier, or a file-like object (see *Python file-like objects*).
- **mode** – Mode in which to open file; one of ("w", "r", "r+", "a", "w-"). See *Opening & creating files*.
- **driver** – File driver to use; see *File drivers*.
- **libver** – Compatibility bounds; see *Version bounding*.
- **userblock_size** – Size (in bytes) of the user block. If nonzero, must be a power of 2 and at least 512. See *User block*.
- **swmr** – If `True` open the file in single-writer-multiple-reader mode. Only used when mode="r".
- **rdcc_nbytes** – Total size of the raw data chunk cache in bytes. The default size is 1024^2 (1 MiB) per dataset.

- **rdcc_w0** – Chunk preemption policy for all datasets. Default value is 0.75.
- **rdcc_nslots** – Number of chunk slots in the raw data chunk cache for this file. Default value is 521.
- **track_order** – Track dataset/group/attribute creation order under root group if `True`. Default is `h5.get_config().track_order`.
- **fs_strategy** – The file space handling strategy to be used. Only allowed when creating a new file. One of "fsm", "page", "aggregate", "none", or `None` (to use the HDF5 default).
- **fs_persist** – A boolean to indicate whether free space should be persistent or not. Only allowed when creating a new file. The default is False.
- **fs_page_size** – File space page size in bytes. Only use when fs_strategy="page". If `None` use the HDF5 default (4096 bytes).
- **fs_threshold** – The smallest free-space section size that the free space manager will track. Only allowed when creating a new file. The default is 1.
- **page_buf_size** – Page buffer size in bytes. Only allowed for HDF5 files created with fs_strategy="page". Must be a power of two value and greater or equal than the file space page size when creating the file. It is not used by default.
- **min_meta_keep** – Minimum percentage of metadata to keep in the page buffer before allowing pages containing metadata to be evicted. Applicable only if `page_buf_size` is set. Default value is zero.
- **min_raw_keep** – Minimum percentage of raw data to keep in the page buffer before allowing pages containing raw data to be evicted. Applicable only if `page_buf_size` is set. Default value is zero.
- **locking** – The file locking behavior. One of:
  - False (or "false") – Disable file locking
  - True (or "true") – Enable file locking
  - "best-effort" – Enable file locking but ignore some errors
  - None – Use HDF5 defaults

  Warning: The HDF5_USE_FILE_LOCKING environment variable can override this parameter.

  Only available with HDF5 >= 1.12.1 or 1.10.x >= 1.10.7.
- **alignment_threshold** – Together with `alignment_interval`, this property ensures that any file object greater than or equal in size to the alignment threshold (in bytes) will be aligned on an address which is a multiple of alignment interval.
- **alignment_interval** – This property should be used in conjunction with `alignment_threshold`. See the description above. For more details, see *Data alignment*.
- **meta_block_size** – Determines the current minimum size, in bytes, of new metadata block allocations. See *Meta block size*.
- **kwds** – Driver-specific keywords; see *File drivers*.

__bool__()

Check that the file descriptor is valid and the file open:

```
>>> f = h5py.File(filename)
>>> f.close()
>>> if f:
...     print("file is open")
... else:
...     print("file is closed")
file is closed
```

close()

Close this file. All open objects will become invalid.

flush()

Request that the HDF5 library flush its buffers to disk.

id

Low-level identifier (an instance of FileID).

filename

Name of this file on disk, as a Unicode string.

mode

String indicating if the file is open readonly ("r") or read-write ("r+"). Will always be one of these two values, regardless of the mode used to open the file.

swmr_mode

True if the file access is using *Single Writer Multiple Reader (SWMR)*. Use mode to distinguish SWMR read from write.

driver

String giving the driver used to open the file. Refer to *File drivers* for a list of drivers.

libver

2-tuple with library version settings. See *Version bounding*.

userblock_size

Size of user block (in bytes). Generally 0. See *User block*.

meta_block_size

Minimum size, in bytes, of metadata block allocations. Default: 2048. See *Meta block size*.

4.2 Groups

Groups are the container mechanism by which HDF5 files are organized. From a Python perspective, they operate somewhat like dictionaries. In this case the "keys" are the names of group members, and the "values" are the members themselves (Group and Dataset) objects.

Group objects also contain most of the machinery which makes HDF5 useful. The *File object* does double duty as the HDF5 *root group*, and serves as your entry point into the file:

```
>>> f = h5py.File('foo.hdf5','w')
>>> f.name
'/'
>>> list(f.keys())
[]
```

Names of all objects in the file are all text strings (`str`). These will be encoded with the HDF5-approved UTF-8 encoding before being passed to the HDF5 C library. Objects may also be retrieved using byte strings, which will be passed on to HDF5 as-is.

4.2.1 Creating groups

New groups are easy to create:

```
>>> grp = f.create_group("bar")
>>> grp.name
'/bar'
>>> subgrp = grp.create_group("baz")
>>> subgrp.name
'/bar/baz'
```

Multiple intermediate groups can also be created implicitly:

```
>>> grp2 = f.create_group("/some/long/path")
>>> grp2.name
'/some/long/path'
>>> grp3 = f['/some/long']
>>> grp3.name
'/some/long'
```

4.2.2 Dict interface and links

Groups implement a subset of the Python dictionary convention. They have methods like `keys()`, `values()` and support iteration. Most importantly, they support the indexing syntax, and standard exceptions:

```
>>> myds = subgrp["MyDS"]
>>> missing = subgrp["missing"]
KeyError: "Name doesn't exist (Symbol table: Object not found)"
```

Objects can be deleted from the file using the standard syntax:

```
>>> del subgroup["MyDataset"]
```

Note: When using h5py from Python 3, the keys(), values() and items() methods will return view-like objects instead of lists. These objects support membership testing and iteration, but can't be sliced like lists.

By default, objects inside group are iterated in alphanumeric order. However, if group is created with `track_order=True`, the insertion order for the group is remembered (tracked) in HDF5 file, and group contents are iterated in that order. The latter is consistent with Python 3.7+ dictionaries.

The default `track_order` for all new groups can be specified globally with `h5.get_config().track_order`.

Hard links

What happens when assigning an object to a name in the group? It depends on the type of object being assigned. For NumPy arrays or other data, the default is to create an *HDF5 datasets*:

```
>>> grp["name"] = 42
>>> out = grp["name"]
>>> out
<HDF5 dataset "name": shape (), type "<i8">
```

When the object being stored is an existing Group or Dataset, a new link is made to the object:

```
>>> grp["other name"] = out
>>> grp["other name"]
<HDF5 dataset "other name": shape (), type "<i8">
```

Note that this is *not* a copy of the dataset! Like hard links in a UNIX file system, objects in an HDF5 file can be stored in multiple groups:

```
>>> grp["other name"] == grp["name"]
True
```

Soft links

Also like a UNIX filesystem, HDF5 groups can contain "soft" or symbolic links, which contain a text path instead of a pointer to the object itself. You can easily create these in h5py by using `h5py.SoftLink`:

```
>>> myfile = h5py.File('foo.hdf5','w')
>>> group = myfile.create_group("somegroup")
>>> myfile["alias"] = h5py.SoftLink('/somegroup')
```

If the target is removed, they will "dangle":

```
>>> del myfile['somegroup']
>>> print(myfile['alias'])
KeyError: 'Component not found (Symbol table: Object not found)'
```

External links

External links are "soft links plus", which allow you to specify the name of the file as well as the path to the desired object. You can refer to objects in any file you wish. Use similar syntax as for soft links:

```
>>> myfile = h5py.File('foo.hdf5','w')
>>> myfile['ext link'] = h5py.ExternalLink("otherfile.hdf5", "/path/to/resource")
```

When the link is accessed, the file "otherfile.hdf5" is opened, and object at "/path/to/resource" is returned.

Since the object retrieved is in a different file, its ".file" and ".parent" properties will refer to objects in that file, *not* the file in which the link resides.

Note: Currently, you can't access an external link if the file it points to is already open. This is related to how HDF5 manages file permissions internally.

Note: The filename is stored in the file as bytes, normally UTF-8 encoded. In most cases, this should work reliably, but problems are possible if a file created on one platform is accessed on another. Older versions of HDF5 may have problems on Windows in particular. See *Filenames on different systems* for more details.

4.2.3 Reference

class h5py.**Group**(*identifier*)

Generally Group objects are created by opening objects in the file, or by the method *Group.create_group()*. Call the constructor with a GroupID instance to create a new Group bound to an existing low-level identifier.

__iter__()

Iterate over the names of objects directly attached to the group. Use *Group.visit()* or *Group.visititems()* for recursive access to group members.

__contains__(*name*)

Dict-like membership testing. *name* may be a relative or absolute path.

__getitem__(*name*)

Retrieve an object. *name* may be a relative or absolute path, or an *object or region reference*. See *Dict interface and links*.

__setitem__(*name*, *value*)

Create a new link, or automatically create a dataset. See *Dict interface and links*.

__bool__()

Check that the group is accessible. A group could be inaccessible for several reasons. For instance, the group, or the file it belongs to, may have been closed elsewhere.

```
>>> f = h5py.open(filename)
>>> group = f["MyGroup"]
>>> f.close()
>>> if group:
...     print("group is accessible")
... else:
...     print("group is inaccessible")
group is inaccessible
```

keys()

Get the names of directly attached group members. Use *Group.visit()* or *Group.visititems()* for recursive access to group members.

Returns
set-like object.

values()

Get the objects contained in the group (Group and Dataset instances). Broken soft or external links show up as None.

Returns
a collection or bag-like object.

items()

Get `(name, value)` pairs for object directly attached to this group. Values for broken soft or external links show up as None.

Returns
a set-like object.

get(*name*, *default=None*, *getclass=False*, *getlink=False*)

Retrieve an item, or information about an item. *name* and *default* work like the standard Python `dict.get`.

Parameters

- **name** – Name of the object to retrieve. May be a relative or absolute path.
- **default** – If the object isn't found, return this instead.
- **getclass** – If True, return the class of object instead; *Group* or *Dataset*.
- **getlink** – If true, return the type of link via a *HardLink*, *SoftLink* or *ExternalLink* instance. If `getclass` is also True, returns the corresponding Link class without instantiating it.

visit(*callable*)

Recursively visit all objects in this group and subgroups. You supply a callable with the signature:

```
callable(name) -> None or return value
```

name will be the name of the object relative to the current group. Return None to continue visiting until all objects are exhausted. Returning anything else will immediately stop visiting and return that value from `visit`:

```
>>> def find_foo(name):
...     """ Find first object with 'foo' anywhere in the name """
...     if 'foo' in name:
...         return name
>>> group.visit(find_foo)
'some/subgroup/foo'
```

visititems(*callable*)

Recursively visit all objects in this group and subgroups. Like *Group.visit()*, except your callable should have the signature:

```
callable(name, object) -> None or return value
```

In this case *object* will be a *Group* or *Dataset* instance.

move(*source*, *dest*)

Move an object or link in the file. If *source* is a hard link, this effectively renames the object. If a soft or external link, the link itself is moved.

Parameters

- **source** (*String*) – Name of object or link to move.
- **dest** (*String*) – New location for object or link.

copy(*source*, *dest*, *name=None*, *shallow=False*, *expand_soft=False*, *expand_external=False*, *expand_refs=False*, *without_attrs=False*)

Copy an object or group. The source can be a path, Group, Dataset, or Datatype object. The destination can be either a path or a Group object. The source and destination need not be in the same file.

If the source is a Group object, by default all objects within that group will be copied recursively.

When the destination is a Group object, by default the target will be created in that group with its current name (basename of obj.name). You can override that by setting "name" to a string.

Parameters

- **source** – What to copy. May be a path in the file or a Group/Dataset object.
- **dest** – Where to copy it. May be a path or Group object.
- **name** – If the destination is a Group object, use this for the name of the copied object (default is basename).
- **shallow** – Only copy immediate members of a group.
- **expand_soft** – Expand soft links into new objects.
- **expand_external** – Expand external links into new objects.
- **expand_refs** – Copy objects which are pointed to by references.
- **without_attrs** – Copy object(s) without copying HDF5 attributes.

create_group(*name*, *track_order=None*)

Create and return a new group in the file.

Parameters

- **name** (*String or None*) – Name of group to create. May be an absolute or relative path. Provide None to create an anonymous group, to be linked into the file later.
- **track_order** – Track dataset/group/attribute creation order under this group if `True`. Default is `h5.get_config().track_order`.

Returns

The new *Group* object.

require_group(*name*)

Open a group in the file, creating it if it doesn't exist. TypeError is raised if a conflicting object already exists. Parameters as in *Group.create_group()*.

create_dataset(*name*, *shape=None*, *dtype=None*, *data=None*, ***kwds*)

Create a new dataset. Options are explained in *Creating datasets*.

Parameters

- **name** – Name of dataset to create. May be an absolute or relative path. Provide None to create an anonymous dataset, to be linked into the file later.
- **shape** – Shape of new dataset (Tuple).
- **dtype** – Data type for new dataset
- **data** – Initialize dataset to this (NumPy array).
- **chunks** – Chunk shape, or True to enable auto-chunking.
- **maxshape** – Dataset will be resizable up to this shape (Tuple). Automatically enables chunking. Use None for the axes you want to be unlimited.
- **compression** – Compression strategy. See *Filter pipeline*.
- **compression_opts** – Parameters for compression filter.
- **scaleoffset** – See *Scale-Offset filter*.

- **shuffle** – Enable shuffle filter (T/**F**). See *Shuffle filter*.
- **fletcher32** – Enable Fletcher32 checksum (T/**F**). See *Fletcher32 filter*.
- **fillvalue** – This value will be used when reading uninitialized parts of the dataset.
- **track_times** – Enable dataset creation timestamps (**T**/F).
- **track_order** – Track attribute creation order if `True`. Default is `h5.get_config().track_order`.
- **external** – Store the dataset in one or more external, non-HDF5 files. This should be an iterable (such as a list) of tuples of `(name, offset, size)` to store data from `offset` to `offset + size` in the named file. Each name must be a str, bytes, or os.PathLike; each offset and size, an integer. The last file in the sequence may have size `h5py.h5f.UNLIMITED` to let it grow as needed. If only a name is given instead of an iterable of tuples, it is equivalent to `[(name, 0, h5py.h5f.UNLIMITED)]`.
- **allow_unknown_filter** – Do not check that the requested filter is available for use (T/F). This should only be set if you will write any data with `write_direct_chunk`, compressing the data before passing it to h5py.
- **rdcc_nbytes** – Total size of the dataset's chunk cache in bytes. The default size is 1024**2 (1 MiB).
- **rdcc_w0** – The chunk preemption policy for this dataset. This must be between 0 and 1 inclusive and indicates the weighting according to which chunks which have been fully read or written are penalized when determining which chunks to flush from cache. A value of 0 means fully read or written chunks are treated no differently than other chunks (the preemption is strictly LRU) while a value of 1 means fully read or written chunks are always preempted before other chunks. If your application only reads or writes data once, this can be safely set to 1. Otherwise, this should be set lower depending on how often you re-read or re-write the same data. The default value is 0.75.
- **rdcc_nslots** – The number of chunk slots in the dataset's chunk cache. Increasing this value reduces the number of cache collisions, but slightly increases the memory used. Due to the hashing strategy, this value should ideally be a prime number. As a rule of thumb, this value should be at least 10 times the number of chunks that can fit in rdcc_nbytes bytes. For maximum performance, this value should be set approximately 100 times that number of chunks. The default value is 521.

require_dataset(*name*, *shape*, *dtype*, *exact=False*, ***kwds*)

Open a dataset, creating it if it doesn't exist.

If keyword "exact" is False (default), an existing dataset must have the same shape and a conversion-compatible dtype to be returned. If True, the shape and dtype must match exactly.

If keyword "maxshape" is given, the maxshape and dtype must match instead.

If any of the keywords "rdcc_nslots", "rdcc_nbytes", or "rdcc_w0" are given, they will be used to configure the dataset's chunk cache.

Other dataset keywords (see create_dataset) may be provided, but are only used if a new dataset is to be created.

Raises TypeError if an incompatible object already exists, or if the shape, maxshape or dtype don't match according to the above rules.

Parameters

exact – Require shape and type to match exactly (T/**F**)

create_dataset_like(*name*, *other*, ***kwds*)

Create a dataset similar to *other*, much like numpy's *_like* functions.

Parameters

- **name** – Name of the dataset (absolute or relative). Provide None to make an anonymous dataset.
- **other** – The dataset whom the new dataset should mimic. All properties, such as shape, dtype, chunking, … will be taken from it, but no data or attributes are being copied.

Any dataset keywords (see create_dataset) may be provided, including shape and dtype, in which case the provided values take precedence over those from *other*.

create_virtual_dataset(*name*, *layout*, *fillvalue=None*)

Create a new virtual dataset in this group. See *Virtual Datasets (VDS)* for more details.

Parameters

- **name** (*str*) – Name of the dataset (absolute or relative).
- **layout** (VirtualLayout) – Defines what source data fills which parts of the virtual dataset.
- **fillvalue** – The value to use where there is no data.

build_virtual_dataset()

Assemble a virtual dataset in this group.

This is used as a context manager:

```
with f.build_virtual_dataset('virt', (10, 1000), np.uint32) as layout:
    layout[0] = h5py.VirtualSource('foo.h5', 'data', (1000,))
```

Inside the context, you populate a *VirtualLayout* object. The file is only modified when you leave the context, and if there's no error.

Parameters

- **name** (*str*) – Name of the dataset (absolute or relative)
- **shape** (*tuple*) – Shape of the dataset
- **dtype** – A numpy dtype for data read from the virtual dataset
- **maxshape** (*tuple*) – Maximum dimensions if the dataset can grow (optional). Use None for unlimited dimensions.
- **fillvalue** – The value used where no data is available.

attrs

Attributes for this group.

id

The groups's low-level identifier; an instance of GroupID.

ref

An HDF5 object reference pointing to this group. See *Using object references*.

regionref

A proxy object allowing you to interrogate region references. See *Using region references*.

name

String giving the full path to this group.

file

File instance in which this group resides.

parent

Group instance containing this group.

4.2.4 Link classes

class h5py.**HardLink**

Exists only to support *Group.get()*. Has no state and provides no properties or methods.

class h5py.**SoftLink**(*path*)

Exists to allow creation of soft links in the file. See *Soft links*. These only serve as containers for a path; they are not related in any way to a particular file.

Parameters
path (*String*) – Value of the soft link.

path

Value of the soft link

class h5py.**ExternalLink**(*filename*, *path*)

Like *SoftLink*, only they specify a filename in addition to a path. See *External links*.

Parameters

- **filename** (*String*) – Name of the file to which the link points
- **path** (*String*) – Path to the object in the external file.

filename

Name of the external file as a Unicode string

path

Path to the object in the external file

4.3 Datasets

Datasets are very similar to NumPy arrays. They are homogeneous collections of data elements, with an immutable datatype and (hyper)rectangular shape. Unlike NumPy arrays, they support a variety of transparent storage features such as compression, error-detection, and chunked I/O.

They are represented in h5py by a thin proxy class which supports familiar NumPy operations like slicing, along with a variety of descriptive attributes:

- **shape** attribute
- **size** attribute
- **ndim** attribute
- **dtype** attribute
- **nbytes** attribute

h5py supports most NumPy dtypes, and uses the same character codes (e.g. `'f'`, `'i8'`) and dtype machinery as Numpy. See *FAQ* for the list of dtypes h5py supports.

4.3.1 Creating datasets

New datasets are created using either *Group.create_dataset()* or *Group.require_dataset()*. Existing datasets should be retrieved using the group indexing syntax (`dset = group["name"]`).

To initialise a dataset, all you have to do is specify a name, shape, and optionally the data type (defaults to `'f'`):

```
>>> dset = f.create_dataset("default", (100,))
>>> dset = f.create_dataset("ints", (100,), dtype='i8')
```

Note: This is not the same as creating an *Empty dataset*.

You may also initialize the dataset to an existing NumPy array by providing the *data* parameter:

```
>>> arr = np.arange(100)
>>> dset = f.create_dataset("init", data=arr)
```

Keywords `shape` and `dtype` may be specified along with `data`; if so, they will override `data.shape` and `data.dtype`. It's required that (1) the total number of points in `shape` match the total number of points in `data.shape`, and that (2) it's possible to cast `data.dtype` to the requested `dtype`.

4.3.2 Reading & writing data

HDF5 datasets reuse the NumPy slicing syntax to read and write to the file. Slice specifications are translated directly to HDF5 "hyperslab" selections, and are a fast and efficient way to access data in the file. The following slicing arguments are recognized:

- Indices: anything that can be converted to a Python long
- Slices (i.e. `[:]` or `[0:10]`)
- Field names, in the case of compound data
- At most one `Ellipsis` (`...`) object
- An empty tuple (`()`) to retrieve all data or *scalar* data

Here are a few examples (output omitted).

```
>>> dset = f.create_dataset("MyDataset", (10,10,10), 'f')
>>> dset[0,0,0]
>>> dset[0,2:10,1:9:3]
>>> dset[:,::2,5]
>>> dset[0]
>>> dset[1,5]
>>> dset[0,...]
>>> dset[...,6]
>>> dset[()]
```

There's more documentation on what parts of numpy's *fancy indexing* are available in h5py.

For compound data, it is advised to separate field names from the numeric slices:

```
>>> dset.fields("FieldA")[:10]   # Read a single field
>>> dset[:10]["FieldA"]          # Read all fields, select in NumPy
```

It is also possible to mix indexing and field names (`dset[:10, "FieldA"]`), but this might be removed in a future version of h5py.

To retrieve the contents of a *scalar* dataset, you can use the same syntax as in NumPy: `result = dset[()]`. In other words, index into the dataset using an empty tuple.

For simple slicing, broadcasting is supported:

```
>>> dset[0,:,:] = np.arange(10)  # Broadcasts to (10,10)
```

Broadcasting is implemented using repeated hyperslab selections, and is safe to use with very large target selections. It is supported for the above "simple" (integer, slice and ellipsis) slicing only.

Warning: Currently h5py does not support nested compound types, see GH1197 for more information.

Multiple indexing

Indexing a dataset once loads a numpy array into memory. If you try to index it twice to write data, you may be surprised that nothing seems to have happened:

```
>>> f = h5py.File('my_hdf5_file.h5', 'w')
>>> dset = f.create_dataset("test", (2, 2))
>>> dset[0][1] = 3.0  # No effect!
>>> print(dset[0][1])
0.0
```

The assignment above only modifies the loaded array. It's equivalent to this:

```
>>> new_array = dset[0]
>>> new_array[1] = 3.0
>>> print(new_array[1])
3.0
>>> print(dset[0][1])
0.0
```

To write to the dataset, combine the indexes in a single step:

```
>>> dset[0, 1] = 3.0
>>> print(dset[0, 1])
3.0
```

Length and iteration

As with NumPy arrays, the `len()` of a dataset is the length of the first axis, and iterating over a dataset iterates over the first axis. However, modifications to the yielded data are not recorded in the file. Resizing a dataset while iterating has undefined results.

On 32-bit platforms, `len(dataset)` will fail if the first axis is bigger than 2**32. It's recommended to use *Dataset.len()* for large datasets.

4.3.3 Chunked storage

An HDF5 dataset created with the default settings will be *contiguous*; in other words, laid out on disk in traditional C order. Datasets may also be created using HDF5's *chunked* storage layout. This means the dataset is divided up into regularly-sized pieces which are stored haphazardly on disk, and indexed using a B-tree.

Chunked storage makes it possible to resize datasets, and because the data is stored in fixed-size chunks, to use compression filters.

To enable chunked storage, set the keyword `chunks` to a tuple indicating the chunk shape:

```
>>> dset = f.create_dataset("chunked", (1000, 1000), chunks=(100, 100))
```

Data will be read and written in blocks with shape (100,100); for example, the data in `dset[0:100,0:100]` will be stored together in the file, as will the data points in range `dset[400:500, 100:200]`.

Chunking has performance implications. It's recommended to keep the total size of your chunks between 10 KiB and 1 MiB, larger for larger datasets. Also keep in mind that when any element in a chunk is accessed, the entire chunk is read from disk.

Since picking a chunk shape can be confusing, you can have h5py guess a chunk shape for you:

```
>>> dset = f.create_dataset("autochunk", (1000, 1000), chunks=True)
```

Auto-chunking is also enabled when using compression or `maxshape`, etc., if a chunk shape is not manually specified.

The iter_chunks method returns an iterator that can be used to perform chunk by chunk reads or writes:

```
>>> for s in dset.iter_chunks():
>>>     arr = dset[s]  # get numpy array for chunk
```

4.3.4 Resizable datasets

In HDF5, datasets can be resized once created up to a maximum size, by calling *Dataset.resize()*. You specify this maximum size when creating the dataset, via the keyword `maxshape`:

```
>>> dset = f.create_dataset("resizable", (10,10), maxshape=(500, 20))
```

Any (or all) axes may also be marked as "unlimited", in which case they may be increased up to the HDF5 per-axis limit of 2**64 elements. Indicate these axes using `None`:

```
>>> dset = f.create_dataset("unlimited", (10, 10), maxshape=(None, 10))
```

Note: Resizing an array with existing data works differently than in NumPy; if any axis shrinks, the data in the missing region is discarded. Data does not "rearrange" itself as it does when resizing a NumPy array.

4.3.5 Filter pipeline

Chunked data may be transformed by the HDF5 *filter pipeline*. The most common use is applying transparent compression. Data is compressed on the way to disk, and automatically decompressed when read. Once the dataset is created with a particular compression filter applied, data may be read and written as normal with no special steps required.

Enable compression with the `compression` keyword to *Group.create_dataset()*:

```
>>> dset = f.create_dataset("zipped", (100, 100), compression="gzip")
```

Options for each filter may be specified with `compression_opts`:

```
>>> dset = f.create_dataset("zipped_max", (100, 100), compression="gzip", compression_
↪opts=9)
```

Lossless compression filters

GZIP filter (`"gzip"`)
Available with every installation of HDF5, so it's best where portability is required. Good compression, moderate speed. `compression_opts` sets the compression level and may be an integer from 0 to 9, default is 4.

LZF filter (`"lzf"`)
Available with every installation of h5py (C source code also available). Low to moderate compression, very fast. No options.

SZIP filter (`"szip"`)
Patent-encumbered filter used in the NASA community. Not available with all installations of HDF5 due to legal reasons. Consult the HDF5 docs for filter options.

Custom compression filters

In addition to the compression filters listed above, compression filters can be dynamically loaded by the underlying HDF5 library. This is done by passing a filter number to *Group.create_dataset()* as the `compression` parameter. The `compression_opts` parameter will then be passed to this filter.

See also:

hdf5plugin
A Python package of several popular filters, including Blosc, LZ4 and ZFP, for convenient use with h5py

HDF5 Filter Plugins
A collection of filters as a single download from The HDF Group

Registered filter plugins
The index of publicly announced filter plugins

Note: The underlying implementation of the compression filter will have the `H5Z_FLAG_OPTIONAL` flag set. This indicates that if the compression filter doesn't compress a block while writing, no error will be thrown. The filter will then be skipped when subsequently reading the block.

Scale-Offset filter

Filters enabled with the `compression` keywords are *lossless*; what comes out of the dataset is exactly what you put in. HDF5 also includes a lossy filter which trades precision for storage space.

Works with integer and floating-point data only. Enable the scale-offset filter by setting *Group.create_dataset()* keyword `scaleoffset` to an integer.

For integer data, this specifies the number of bits to retain. Set to 0 to have HDF5 automatically compute the number of bits required for lossless compression of the chunk. For floating-point data, indicates the number of digits after the decimal point to retain.

Warning: Currently the scale-offset filter does not preserve special float values (i.e. NaN, inf), see https://forum.hdfgroup.org/t/scale-offset-filter-and-special-float-values-nan-infinity/3379 for more information and follow-up.

Shuffle filter

Block-oriented compressors like GZIP or LZF work better when presented with runs of similar values. Enabling the shuffle filter rearranges the bytes in the chunk and may improve compression ratio. No significant speed penalty, lossless.

Enable by setting *Group.create_dataset()* keyword `shuffle` to True.

Fletcher32 filter

Adds a checksum to each chunk to detect data corruption. Attempts to read corrupted chunks will fail with an error. No significant speed penalty. Obviously shouldn't be used with lossy compression filters.

Enable by setting *Group.create_dataset()* keyword `fletcher32` to True.

4.3.6 Multi-Block Selection

The full H5Sselect_hyperslab API is exposed via the MultiBlockSlice object. This takes four elements to define the selection (start, count, stride and block) in contrast to the built-in slice object, which takes three elements. A MultiBlockSlice can be used in place of a slice to select a number of (count) blocks of multiple elements separated by a stride, rather than a set of single elements separated by a step.

For an explanation of how this slicing works, see the HDF5 documentation.

For example:

```
>>> dset[...]
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> dset[MultiBlockSlice(start=1, count=3, stride=4, block=2)]
array([ 1,  2,  5,  6,  9, 10])
```

They can be used in multi-dimensional slices alongside any slicing object, including other MultiBlockSlices. For a more complete example of this, see the multiblockslice_interleave.py example script.

4.3.7 Fancy indexing

A subset of the NumPy fancy-indexing syntax is supported. Use this with caution, as the underlying HDF5 mechanisms may have different performance than you expect.

For any axis, you can provide an explicit list of points you want; for a dataset with shape (10, 10):

```
>>> dset.shape
(10, 10)
>>> result = dset[0, [1,3,8]]
>>> result.shape
(3,)
>>> result = dset[1:6, [5,8,9]]
>>> result.shape
(5, 3)
```

The following restrictions exist:

- Selection coordinates must be given in increasing order
- Duplicate selections are ignored
- Very long lists (> 1000 elements) may produce poor performance

NumPy boolean "mask" arrays can also be used to specify a selection. The result of this operation is a 1-D array with elements arranged in the standard NumPy (C-style) order. Behind the scenes, this generates a laundry list of points to select, so be careful when using it with large masks:

```
>>> arr = numpy.arange(100).reshape((10,10))
>>> dset = f.create_dataset("MyDataset", data=arr)
>>> result = dset[arr > 50]
>>> result.shape
(49,)
```

Changed in version 2.10: Selecting using an empty list is now allowed. This returns an array with length 0 in the relevant dimension.

4.3.8 Creating and Reading Empty (or Null) datasets and attributes

HDF5 has the concept of Empty or Null datasets and attributes. These are not the same as an array with a shape of (), or a scalar dataspace in HDF5 terms. Instead, it is a dataset with an associated type, no data, and no shape. In h5py, we represent this as either a dataset with shape `None`, or an instance of `h5py.Empty`. Empty datasets and attributes cannot be sliced.

To create an empty attribute, use `h5py.Empty` as per *Attributes*:

```
>>> obj.attrs["EmptyAttr"] = h5py.Empty("f")
```

Similarly, reading an empty attribute returns `h5py.Empty`:

```
>>> obj.attrs["EmptyAttr"]
h5py.Empty(dtype="f")
```

Empty datasets can be created either by defining a `dtype` but no `shape` in `create_dataset`:

```
>>> grp.create_dataset("EmptyDataset", dtype="f")
```

or by `data` to an instance of `h5py.Empty`:

```
>>> grp.create_dataset("EmptyDataset", data=h5py.Empty("f"))
```

An empty dataset has shape defined as `None`, which is the best way of determining whether a dataset is empty or not. An empty dataset can be "read" in a similar way to scalar datasets, i.e. if `empty_dataset` is an empty dataset:

```
>>> empty_dataset[()]
h5py.Empty(dtype="f")
```

The dtype of the dataset can be accessed via `<dset>.dtype` as per normal. As empty datasets cannot be sliced, some methods of datasets such as `read_direct` will raise a `TypeError` exception if used on a empty dataset.

4.3.9 Reference

class h5py.**Dataset**(*identifier*)

Dataset objects are typically created via *Group.create_dataset()*, or by retrieving existing datasets from a file. Call this constructor to create a new Dataset bound to an existing DatasetID identifier.

__getitem__(*args*)

NumPy-style slicing to retrieve data. See *Reading & writing data*.

__setitem__(*args*)

NumPy-style slicing to write data. See *Reading & writing data*.

__bool__()

Check that the dataset is accessible. A dataset could be inaccessible for several reasons. For instance, the dataset, or the file it belongs to, may have been closed elsewhere.

```
>>> f = h5py.open(filename)
>>> dset = f["MyDS"]
>>> f.close()
>>> if dset:
...     print("dataset accessible")
... else:
...     print("dataset inaccessible")
dataset inaccessible
```

read_direct(*array*, *source_sel=None*, *dest_sel=None*)

Read from an HDF5 dataset directly into a NumPy array, which can avoid making an intermediate copy as happens with slicing. The destination array must be C-contiguous and writable, and must have a datatype to which the source data may be cast. Data type conversion will be carried out on the fly by HDF5.

source_sel and *dest_sel* indicate the range of points in the dataset and destination array respectively. Use the output of `numpy.s_[args]`:

```
>>> dset = f.create_dataset("dset", (100,), dtype='int64')
>>> arr = np.zeros((100,), dtype='int32')
>>> dset.read_direct(arr, np.s_[0:10], np.s_[50:60])
```

write_direct(*source*, *source_sel=None*, *dest_sel=None*)

Write data directly to HDF5 from a NumPy array. The source array must be C-contiguous. Selections must be the output of numpy.s_[<args>]. Broadcasting is supported for simple indexing.

astype(*dtype*)

Return a wrapper allowing you to read data as a particular type. Conversion is handled by HDF5 directly, on the fly:

```
>>> dset = f.create_dataset("bigint", (1000,), dtype='int64')
>>> out = dset.astype('int16')[:]
>>> out.dtype
dtype('int16')
```

Changed in version 3.9: *astype()* can no longer be used as a context manager.

asstr(*encoding=None*, *errors='strict'*)

Only for string datasets. Returns a wrapper to read data as Python string objects:

```
>>> s = dataset.asstr()[0]
```

encoding and errors work like `bytes.decode()`, but the default encoding is defined by the datatype - ASCII or UTF-8. This is not guaranteed to be correct.

New in version 3.0.

fields(*names*)

Get a wrapper to read a subset of fields from a compound data type:

```
>>> 2d_coords = dataset.fields(['x', 'y'])[:]
```

If names is a string, a single field is extracted, and the resulting arrays will have that dtype. Otherwise, it should be an iterable, and the read data will have a compound dtype.

New in version 3.0.

iter_chunks()

Iterate over chunks in a chunked dataset. The optional `sel` argument is a slice or tuple of slices that defines the region to be used. If not set, the entire dataspace will be used for the iterator.

For each chunk within the given region, the iterator yields a tuple of slices that gives the intersection of the given chunk with the selection area. This can be used to *read or write data in that chunk*.

A TypeError will be raised if the dataset is not chunked.

A ValueError will be raised if the selection region is invalid.

New in version 3.0.

resize(*size*, *axis=None*)

Change the shape of a dataset. *size* may be a tuple giving the new dataset shape, or an integer giving the new length of the specified *axis*.

Datasets may be resized only up to *Dataset.maxshape*.

len()

Return the size of the first axis.

make_scale(*name=''*)

Make this dataset an HDF5 *dimension scale*.

You can then attach it to dimensions of other datasets like this:

```
other_ds.dims[0].attach_scale(ds)
```

You can optionally pass a name to associate with this scale.

virtual_sources()

If this dataset is a *virtual dataset*, return a list of named tuples: `(vspace, file_name, dset_name, src_space)`, describing which parts of the dataset map to which source datasets. The two 'space' members are low-level `SpaceID` objects.

shape

NumPy-style shape tuple giving dataset dimensions.

dtype

NumPy dtype object giving the dataset's type.

size

Integer giving the total number of elements in the dataset.

nbytes

Integer giving the total number of bytes required to load the full dataset into RAM (i.e. *dset[()]*). This may not be the amount of disk space occupied by the dataset, as datasets may be compressed when written or only partly filled with data. This value also does not include the array overhead, as it only describes the size of the data itself. Thus the real amount of RAM occupied by this dataset may be slightly greater.

New in version 3.0.

ndim

Integer giving the total number of dimensions in the dataset.

maxshape

NumPy-style shape tuple indicating the maximum dimensions up to which the dataset may be resized. Axes with `None` are unlimited.

chunks

Tuple giving the chunk shape, or None if chunked storage is not used. See *Chunked storage*.

compression

String with the currently applied compression filter, or None if compression is not enabled for this dataset. See *Filter pipeline*.

compression_opts

Options for the compression filter. See *Filter pipeline*.

scaleoffset

Setting for the HDF5 scale-offset filter (integer), or None if scale-offset compression is not used for this dataset. See *Scale-Offset filter*.

shuffle

Whether the shuffle filter is applied (T/F). See *Shuffle filter*.

fletcher32

Whether Fletcher32 checksumming is enabled (T/F). See *Fletcher32 filter*.

fillvalue

Value used when reading uninitialized portions of the dataset, or None if no fill value has been defined, in which case HDF5 will use a type-appropriate default value. Can't be changed after the dataset is created.

external

If this dataset is stored in one or more external files, this is a list of 3-tuples, like the `external=` parameter to `Group.create_dataset()`. Otherwise, it is `None`.

is_virtual

True if this dataset is a *virtual dataset*, otherwise False.

dims

Access to *Dimension Scales*.

is_scale

Return `True` if the dataset is also a *dimension scale*, `False` otherwise.

attrs

Attributes for this dataset.

id

The dataset's low-level identifier; an instance of `DatasetID`.

ref

An HDF5 object reference pointing to this dataset. See *Using object references*.

regionref

Proxy object for creating HDF5 region references. See *Using region references*.

name

String giving the full path to this dataset.

file

`File` instance in which this dataset resides

parent

`Group` instance containing this dataset.

4.4 Attributes

Attributes are a critical part of what makes HDF5 a "self-describing" format. They are small named pieces of data attached directly to *`Group`* and *`Dataset`* objects. This is the official way to store metadata in HDF5.

Each Group or Dataset has a small proxy object attached to it, at `<obj>.attrs`. Attributes have the following properties:

- They may be created from any scalar or NumPy array
- Each attribute should be small (generally < 64k)
- There is no partial I/O (i.e. slicing); the entire attribute must be read.

The `.attrs` proxy objects are of class *`AttributeManager`*, below. This class supports a dictionary-style interface.

By default, attributes are iterated in alphanumeric order. However, if group or dataset is created with `track_order=True`, the attribute insertion order is remembered (tracked) in HDF5 file, and iteration uses that order. The latter is consistent with Python 3.7+ dictionaries.

The default `track_order` for all new groups and datasets can be specified globally with `h5.get_config().track_order`.

4.4.1 Large attributes

HDF5 allows attributes to be larger than 64 KiB, but these need to be stored in a different way. As of March 2024, the way HDF5 documentation suggests you configure this does not work. Instead, enable order tracking when creating the object you want to attach attributes to:

```
grp = f.create_group('g', track_order=True)
grp.attrs['large'] = np.arange(1_000_000, dtype=np.uint32)
```

4.4.2 Reference

class h5py.**AttributeManager**(*parent*)

AttributeManager objects are created directly by h5py. You should access instances by `group.attrs` or `dataset.attrs`, not by manually creating them.

__iter__()

Get an iterator over attribute names.

__contains__(*name*)

Determine if attribute *name* is attached to this object.

__getitem__(*name*)

Retrieve an attribute.

__setitem__(*name*, *val*)

Create an attribute, overwriting any existing attribute. The type and shape of the attribute are determined automatically by h5py.

__delitem__(*name*)

Delete an attribute. KeyError if it doesn't exist.

keys()

Get the names of all attributes attached to this object.

Returns
set-like object.

values()

Get the values of all attributes attached to this object.

Returns
collection or bag-like object.

items()

Get `(name, value)` tuples for all attributes attached to this object.

Returns
collection or set-like object.

get(*name*, *default=None*)

Retrieve *name*, or *default* if no such attribute exists.

get_id(*name*)

Get the low-level `AttrID` for the named attribute.

create(*name*, *data*, *shape=None*, *dtype=None*)

Create a new attribute, with control over the shape and type. Any existing attribute will be overwritten.

Parameters

- **name** (*String*) – Name of the new attribute
- **data** – Value of the attribute; will be put through `numpy.array(data)`.
- **shape** (*Tuple*) – Shape of the attribute. Overrides `data.shape` if both are given, in which case the total number of points must be unchanged.
- **dtype** (*NumPy dtype*) – Data type for the attribute. Overrides `data.dtype` if both are given.

modify(*name*, *value*)

Change the value of an attribute while preserving its type and shape. Unlike *AttributeManager.__setitem__()*, if the attribute already exists, only its value will be changed. This can be useful for interacting with externally generated files, where the type and shape must not be altered.

If the attribute doesn't exist, it will be created with a default shape and type.

Parameters

- **name** (*String*) – Name of attribute to modify.
- **value** – New value. Will be put through `numpy.array(value)`.

4.5 Dimension Scales

Datasets are multidimensional arrays. HDF5 provides support for labeling the dimensions and associating one or more "dimension scales" with each dimension. A dimension scale is simply another HDF5 dataset. In principle, the length of the multidimensional array along the dimension of interest should be equal to the length of the dimension scale, but HDF5 does not enforce this property.

The HDF5 library provides the H5DS API for working with dimension scales. H5py provides low-level bindings to this API in `h5py.h5ds`. These low-level bindings are in turn used to provide a high-level interface through the `Dataset.dims` property. Suppose we have the following data file:

```
f = File('foo.h5', 'w')
f['data'] = np.ones((4, 3, 2), 'f')
```

HDF5 allows the dimensions of `data` to be labeled, for example:

```
f['data'].dims[0].label = 'z'
f['data'].dims[2].label = 'x'
```

Note that the first dimension, which has a length of 4, has been labeled "z", the third dimension (in this case the fastest varying dimension), has been labeled "x", and the second dimension was given no label at all.

We can also use HDF5 datasets as dimension scales. For example, if we have:

```
f['x1'] = [1, 2]
f['x2'] = [1, 1.1]
f['y1'] = [0, 1, 2]
f['z1'] = [0, 1, 4, 9]
```

We are going to treat the `x1`, `x2`, `y1`, and `z1` datasets as dimension scales:

```
f['x1'].make_scale()
f['x2'].make_scale('x2 name')
f['y1'].make_scale('y1 name')
f['z1'].make_scale('z1 name')
```

When you create a dimension scale, you may provide a name for that scale. In this case, the `x1` scale was not given a name, but the others were. Now we can associate these dimension scales with the primary dataset:

```
f['data'].dims[0].attach_scale(f['z1'])
f['data'].dims[1].attach_scale(f['y1'])
f['data'].dims[2].attach_scale(f['x1'])
f['data'].dims[2].attach_scale(f['x2'])
```

Note that two dimension scales were associated with the third dimension of `data`. You can also detach a dimension scale:

```
f['data'].dims[2].detach_scale(f['x2'])
```

but for now, lets assume that we have both `x1` and `x2` still associated with the third dimension of `data`. You can attach a dimension scale to any number of HDF5 datasets, you can even attach it to multiple dimensions of a single HDF5 dataset.

Now that the dimensions of `data` have been labeled, and the dimension scales for the various axes have been specified, we have provided much more context with which `data` can be interpreted. For example, if you want to know the labels for the various dimensions of `data`:

```
>>> [dim.label for dim in f['data'].dims]
['z', '', 'x']
```

If you want the names of the dimension scales associated with the "x" axis:

```
>>> f['data'].dims[2].keys()
['', 'x2 name']
```

`items()` and `values()` methods are also provided. The dimension scales themselves can also be accessed with:

```
f['data'].dims[2][1]
```

or:

```
f['data'].dims[2]['x2 name']
```

such that:

```
>>> f['data'].dims[2][1] == f['x2']
True
```

though, beware that if you attempt to index the dimension scales with a string, the first dimension scale whose name matches the string is the one that will be returned. There is no guarantee that the name of the dimension scale is unique.

Nested dimension scales are not permitted: if a dataset has a dimension scale attached to it, converting the dataset to a dimension scale will fail, since the HDF5 specification doesn't allow this.

```
>>> f['data'].make_scale()
RuntimeError: Unspecified error in H5DSset_scale (return value <0)
```

4.6 Low-Level API

This documentation mostly describes the h5py high-level API, which offers the main features of HDF5 in an interface modelled on dictionaries and NumPy arrays. h5py also provides a low-level API, which more closely follows the HDF5 C API.

See also:

- h5py Low-Level API Reference
- HDF5 C/Fortran Reference Manual

You can easily switch between the two levels in your code:

- **To the low-level**: High-level *File*, *Group* and *Dataset* objects all have a `.id` attribute exposing the corresponding low-level objects—`FileID`, `GroupID` and `DatasetID`:

```
dsid = dset.id
dsid.get_offset()  # Low-level method
```

  Although there is no high-level object for a single attribute, *AttributeManager.get_id()* will get the low-level `AttrID` object:

```
aid = dset.attrs.get_id('timestamp')
aid.get_storage_size()  # Low-level method
```

- **To the high-level**: Low-level `FileID`, `GroupID` and `DatasetID` objects can be passed to the constructors of *File*, *Group* and *Dataset*, respectively.

CHAPTER

FIVE

ADVANCED TOPICS

5.1 Configuring h5py

5.1.1 Library configuration

A few library options are available to change the behavior of the library. You can get a reference to the global library configuration object via the function `h5py.get_config()`. This object supports the following attributes:

complex_names
Set to a 2-tuple of strings (real, imag) to control how complex numbers are saved. The default is ('r','i').

bool_names
Booleans are saved as HDF5 enums. Set this to a 2-tuple of strings (false, true) to control the names used in the enum. The default is ("FALSE", "TRUE").

track_order
Whether to track dataset/group/attribute creation order. If container creation order is tracked, its links and attributes are iterated in ascending creation order (consistent with `dict` in Python 3.7+); otherwise in ascending alphanumeric order. Global configuration value can be overridden for particular container by specifying `track_order` argument to *h5py.File*, *h5py.Group.create_group()*, *h5py.Group.create_dataset()*. The default is `False`.

5.2 Special types

HDF5 supports a few types which have no direct NumPy equivalent. Among the most useful and widely used are *variable-length* (VL) types, and enumerated types. As of version 2.3, h5py fully supports HDF5 enums and VL types.

5.2.1 How special types are represented

Since there is no direct NumPy dtype for variable-length strings, enums or references, h5py extends the dtype system slightly to let HDF5 know how to store these types. Each type is represented by a native NumPy dtype, with a small amount of metadata attached. NumPy routines ignore the metadata, but h5py can use it to determine how to store the data.

The metadata h5py attaches to dtypes is not part of the public API, so it may change between versions. Use the functions described below to create and check for these types.

5.2.2 Variable-length strings

See also:

Strings in HDF5

In HDF5, data in VL format is stored as arbitrary-length vectors of a base type. In particular, strings are stored C-style in null-terminated buffers. NumPy has no native mechanism to support this. Unfortunately, this is the de facto standard for representing strings in the HDF5 C API, and in many HDF5 applications.

Thankfully, NumPy has a generic pointer type in the form of the "object" ("O") dtype. In h5py, variable-length strings are mapped to object arrays. A small amount of metadata attached to an "O" dtype tells h5py that its contents should be converted to VL strings when stored in the file.

Existing VL strings can be read and written to with no additional effort; Python strings and fixed-length NumPy strings can be auto-converted to VL data and stored.

Here's an example showing how to create a VL array of strings:

```
>>> f = h5py.File('foo.hdf5')
>>> dt = h5py.string_dtype(encoding='utf-8')
>>> ds = f.create_dataset('VLDS', (100,100), dtype=dt)
>>> ds.dtype.kind
'O'
>>> h5py.check_string_dtype(ds.dtype)
string_info(encoding='utf-8', length=None)
```

h5py.**string_dtype**(*encoding='utf-8'*, *length=None*)

Make a numpy dtype for HDF5 strings

Parameters

- **encoding** – `'utf-8'` or `'ascii'`.
- **length** – `None` for variable-length, or an integer for fixed-length string data, giving the length in bytes.

h5py.**check_string_dtype**(*dt*)

Check if `dt` is a string dtype. Returns a *string_info* object if it is, or `None` if not.

class h5py.**string_info**

A named tuple type holding string encoding and length.

encoding

The character encoding associated with the string dtype, which can be `'utf-8'` or `'ascii'`.

length

For fixed-length string dtypes, the length in bytes. `None` for variable-length strings.

5.2.3 Arbitrary vlen data

Starting with h5py 2.3, variable-length types are not restricted to strings. For example, you can create a "ragged" array of integers:

```
>>> dt = h5py.vlen_dtype(np.dtype('int32'))
>>> dset = f.create_dataset('vlen_int', (100,), dtype=dt)
>>> dset[0] = [1,2,3]
>>> dset[1] = [1,2,3,4,5]
```

Single elements are read as NumPy arrays:

```
>>> dset[0]
array([1, 2, 3], dtype=int32)
```

Multidimensional selections produce an object array whose members are integer arrays:

```
>>> dset[0:2]
array([array([1, 2, 3], dtype=int32), array([1, 2, 3, 4, 5], dtype=int32)], dtype=object)
```

Note: NumPy doesn't support ragged arrays, and the 'arrays of arrays' h5py uses as a workaround are not as convenient or efficient as regular NumPy arrays. If you're deciding how to store data, consider whether there's a sensible way to do it without a variable-length type.

---

h5py.**vlen_dtype**(*basetype*)

Make a numpy dtype for an HDF5 variable-length datatype.

Parameters

basetype – The dtype of each element in the array.

h5py.**check_vlen_dtype**(*dt*)

Check if `dt` is a variable-length dtype. Returns the base type if it is, or `None` if not.

5.2.4 Enumerated types

HDF5 has the concept of an *enumerated type*, which is an integer datatype with a restriction to certain named values. Since NumPy has no such datatype, HDF5 ENUM types are read and written as integers.

Here's an example of creating an enumerated type:

```
>>> dt = h5py.enum_dtype({"RED": 0, "GREEN": 1, "BLUE": 42}, basetype='i')
>>> h5py.check_enum_dtype(dt)
{'BLUE': 42, 'GREEN': 1, 'RED': 0}
>>> f = h5py.File('foo.hdf5','w')
>>> ds = f.create_dataset("EnumDS", (100,100), dtype=dt)
>>> ds.dtype.kind
'i'
>>> ds[0,:] = 42
>>> ds[0,0]
42
>>> ds[1,0]
0
```

h5py.**enum_dtype**(*values_dict*, *basetype=np.uint8*)

Create a NumPy representation of an HDF5 enumerated type

Parameters

- **values_dict** – Mapping of string names to integer values.
- **basetype** – An appropriate integer base dtype large enough to hold the possible options.

h5py.**check_enum_dtype**(*dt*)

Check if `dt` represents an enumerated type. Returns the values dict if it is, or `None` if not.

5.2.5 Object and region references

References have their *own section*.

5.2.6 Storing other types as opaque data

New in version 3.0.

Numpy datetime64 and timedelta64 dtypes have no equivalent in HDF5 (the HDF5 time type is broken and deprecated). h5py allows you to store such data with an HDF5 opaque type; it can be read back correctly by h5py, but won't be interoperable with other tools.

Here's an example of storing and reading a datetime array:

```
>>> arr = np.array([np.datetime64('2019-09-22T17:38:30')])
>>> f['data'] = arr.astype(h5py.opaque_dtype(arr.dtype))
>>> print(f['data'][:])
['2019-09-22T17:38:30']
```

h5py.**opaque_dtype**(*dt*)

Return a dtype like the input, tagged to be stored as HDF5 opaque type.

h5py.**check_opaque_dtype**(*dt*)

Return True if the dtype given is tagged to be stored as HDF5 opaque data.

Note: With some exceptions, you can use *opaque_dtype()* with any numpy dtype. While this may seem like a convenient way to get arbitrary data into HDF5, remember that it's not a standard format. It's better to fit your data into HDF5's native structures, or use a file format better suited to your data.

5.2.7 Older API

Before h5py 2.10, a single pair of functions was used to create and check for all of these special dtypes. These are still available for backwards compatibility, but are deprecated in favour of the functions listed above.

h5py.**special_dtype**(***kwds*)

Create a NumPy dtype object containing type hints. Only one keyword may be specified.

Parameters

- **vlen** – Base type for HDF5 variable-length datatype.
- **enum** – 2-tuple `(basetype, values_dict)`. `basetype` must be an integer dtype; `values_dict` is a dictionary mapping string names to integer values.

- **ref** – Provide class `h5py.Reference` or `h5py.RegionReference` to create a type representing object or region references respectively.

`h5py.check_dtype`(***kwds*)

Determine if the given dtype object is a special type. Example:

```
>>> out = h5py.check_dtype(vlen=mydtype)
>>> if out is not None:
...     print("Vlen of type %s" % out)
str
```

Parameters

- **vlen** – Check for an HDF5 variable-length type; returns base class
- **enum** – Check for an enumerated type; returns 2-tuple `(basetype, values_dict)`.
- **ref** – Check for an HDF5 object or region reference; returns either `h5py.Reference` or `h5py.RegionReference`.

5.3 Strings in HDF5

Note: The rules around reading & writing string data were redesigned for h5py 3.0. Refer to the h5py 2.10 docs for how to store strings in older versions.

5.3.1 Reading strings

String data in HDF5 datasets is read as bytes by default: `bytes` objects for variable-length strings, or numpy bytes arrays (`'S'` dtypes) for fixed-length strings. Use *`Dataset.asstr()`* to retrieve `str` objects.

Variable-length strings in attributes are read as `str` objects. These are decoded as UTF-8 with surrogate escaping for unrecognised bytes. Fixed-length strings are read as numpy bytes arrays, the same as for datasets.

5.3.2 Storing strings

When creating a new dataset or attribute, Python `str` or `bytes` objects will be treated as variable-length strings, marked as UTF-8 and ASCII respectively. Numpy bytes arrays (`'S'` dtypes) make fixed-length strings. You can use *`string_dtype()`* to explicitly specify any HDF5 string datatype.

When writing data to an existing dataset or attribute, data passed as bytes is written without checking the encoding. Data passed as Python `str` objects is encoded as either ASCII or UTF-8, based on the HDF5 datatype. In either case, null bytes (`'\x00'`) in the data will cause an error.

Warning: Fixed-length string datasets will silently truncate longer strings which are written to them. Numpy byte string arrays do the same thing.

Fixed-length strings in HDF5 hold a set number of bytes. It may take multiple bytes to store one character.

What about NumPy's `U` type?

NumPy also has a Unicode type, a UTF-32 fixed-width format (4-byte characters). HDF5 has no support for wide characters. Rather than trying to hack around this and "pretend" to support it, h5py will raise an error if you try to store data of this type.

5.3.3 How to store raw binary data

If you have a non-text blob in a Python byte string (as opposed to ASCII or UTF-8 encoded text, which is fine), you should wrap it in a `void` type for storage. This will map to the HDF5 OPAQUE datatype, and will prevent your blob from getting mangled by the string machinery.

Here's an example of how to store binary data in an attribute, and then recover it:

```
>>> binary_blob = b"Hello\x00Hello\x00"
>>> dset.attrs["attribute_name"] = np.void(binary_blob)
>>> out = dset.attrs["attribute_name"]
>>> binary_blob = out.tobytes()
```

5.3.4 Object names

Unicode strings are used exclusively for object names in the file:

```
>>> f.name
'/'
```

You can supply either byte or unicode strings when creating or retrieving objects. If a byte string is supplied, it will be used as-is; Unicode strings will be encoded as UTF-8.

In the file, h5py uses the most-compatible representation; H5T_CSET_ASCII for characters in the ASCII range; H5T_CSET_UTF8 otherwise.

```
>>> grp = f.create_dataset(b"name")
>>> grp2 = f.create_dataset("name2")
```

5.3.5 Encodings

HDF5 supports two string encodings: ASCII and UTF-8. We recommend using UTF-8 when creating HDF5 files, and this is what h5py does by default with Python `str` objects. If you need to write ASCII for compatibility reasons, you should ensure you only write pure ASCII characters (this can be done by `your_string.encode("ascii")`), as otherwise your text may turn into mojibake. You can use *`string_dtype()`* to specify the encoding for string data.

See also:

Joel Spolsky's introduction to Unicode & character sets
If this section looks like gibberish, try this.

For reading, as long as the encoding metadata is correct, the defaults for *`Dataset.asstr()`* will always work. However, HDF5 does not enforce the string encoding, and there are files where the encoding metadata doesn't match what's really stored. Most commonly, data marked as ASCII may be in one of the many "Extended ASCII" encodings such as Latin-1. If you know what encoding your data is in, you can specify this using *`Dataset.asstr()`*. If you have data in an unknown encoding, you can also use any of the builtin python error handlers.

Variable-length strings in attributes are read as `str` objects, decoded as UTF-8 with the `'surrogateescape'` error handler. If an attribute is incorrectly encoded, you'll see 'surrogate' characters such as `'\udcb1'` when reading it:

```
>>> s = "2.0±0.1"
>>> f.attrs["string_good"] = s  # Good - h5py uses UTF-8
>>> f.attrs["string_bad"] = s.encode("latin-1")  # Bad!
>>> f.attrs["string_bad"]
'2.0\udcb10.1'
```

To recover the original string, you'll need to *encode* it with UTF-8, and then decode it with the correct encoding:

```
>>> f.attrs["string_bad"].encode('utf-8', 'surrogateescape').decode('latin-1')
'2.0±0.1'
```

Fixed length strings are different; h5py doesn't try to decode them:

```
>>> s = "2.0±0.1"
>>> utf8_type = h5py.string_dtype('utf-8', 30)
>>> ascii_type = h5py.string_dtype('ascii', 30)
>>> f.attrs["fixed_good"] = np.array(s.encode("utf-8"), dtype=utf8_type)
>>> f.attrs["fixed_bad"] = np.array(s.encode("latin-1"), dtype=ascii_type)
>>> f.attrs["fixed_bad"]
b'2.0\xb10.1'
>>> f.attrs["fixed_bad"].decode("utf-8")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    f.attrs["fixed_bad"].decode("utf-8")
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 3: invalid start␣
↪byte
>>> f.attrs["fixed_bad"].decode("latin-1")
'2.0±0.1'
```

As we get bytes back, we only need to decode them with the correct encoding.

5.4 Object and Region References

In addition to soft and external links, HDF5 supplies one more mechanism to refer to objects and data in a file. HDF5 *references* are low-level pointers to other objects. The great advantage of references is that they can be stored and retrieved as data; you can create an attribute or an entire dataset of reference type.

References come in two flavors, object references and region references. As the name suggests, object references point to a particular object in a file, either a dataset, group or named datatype. Region references always point to a dataset, and additionally contain information about a certain selection (*dataset region*) on that dataset. For example, if you have a dataset representing an image, you could specify a region of interest, and store it as an attribute on the dataset.

5.4.1 Using object references

It's trivial to create a new object reference; every high-level object in h5py has a read-only property "ref", which when accessed returns a new object reference:

```
>>> myfile = h5py.File('myfile.hdf5')
>>> mygroup = myfile['/some/group']
>>> ref = mygroup.ref
>>> print(ref)
<HDF5 object reference>
```

"Dereferencing" these objects is straightforward; use the same syntax as when opening any other object:

```
>>> mygroup2 = myfile[ref]
>>> print(mygroup2)
<HDF5 group "/some/group" (0 members)>
```

5.4.2 Using region references

Region references always contain a selection. You create them using the dataset property "regionref" and standard NumPy slicing syntax:

```
>>> myds = myfile.create_dataset('dset', (200,200))
>>> regref = myds.regionref[0:10, 0:5]
>>> print(regref)
<HDF5 region reference>
```

The reference itself can now be used in place of slicing arguments to the dataset:

```
>>> subset = myds[regref]
```

For selections which don't conform to a regular grid, h5py copies the behavior of NumPy's fancy indexing, which returns a 1D array. Note that for h5py release before 2.2, h5py always returns a 1D array.

In addition to storing a selection, region references inherit from object references, and can be used anywhere an object reference is accepted. In this case the object they point to is the dataset used to create them.

5.4.3 Storing references in a dataset

HDF5 treats object and region references as data. Consequently, there is a special HDF5 type to represent them. However, NumPy has no equivalent type. Rather than implement a special "reference type" for NumPy, references are handled at the Python layer as plain, ordinary python objects. To NumPy they are represented with the "object" dtype (kind 'O'). A small amount of metadata attached to the dtype tells h5py to interpret the data as containing reference objects.

These dtypes are available from h5py for references and region references:

- `h5py.ref_dtype` - for object references
- `h5py.regionref_dtype` - for region references

To store an array of references, use the appropriate dtype when creating the dataset:

```
>>> ref_dataset = myfile.create_dataset("MyRefs", (100,), dtype=h5py.ref_dtype)
```

You can read from and write to the array as normal:

```
>>> ref_dataset[0] = myfile.ref
>>> print(ref_dataset[0])
<HDF5 object reference>
```

5.4.4 Storing references in an attribute

Simply assign the reference to a name; h5py will figure it out and store it with the correct type:

```
>>> myref = myfile.ref
>>> myfile.attrs["Root group reference"] = myref
```

5.4.5 Null references

When you create a dataset of reference type, the uninitialized elements are "null" references. H5py uses the truth value of a reference object to indicate whether or not it is null:

```
>>> print(bool(myfile.ref))
True
>>> nullref = ref_dataset[50]
>>> print(bool(nullref))
False
```

5.5 Parallel HDF5

Read-only parallel access to HDF5 files works with no special preparation: each process should open the file independently and read data normally (avoid opening the file and then forking).

Parallel HDF5 is a feature built on MPI which also supports *writing* an HDF5 file in parallel. To use this, both HDF5 and h5py must be compiled with MPI support turned on, as described below.

5.5.1 How does Parallel HDF5 work?

Parallel HDF5 is a configuration of the HDF5 library which lets you share open files across multiple parallel processes. It uses the MPI (Message Passing Interface) standard for interprocess communication. Consequently, when using Parallel HDF5 from Python, your application will also have to use the MPI library.

This is accomplished through the mpi4py Python package, which provides excellent, complete Python bindings for MPI. Here's an example "Hello World" using `mpi4py`:

```
>>> from mpi4py import MPI
>>> print("Hello World (from process %d)" % MPI.COMM_WORLD.Get_rank())
```

To run an MPI-based parallel program, use the `mpiexec` program to launch several parallel instances of Python:

```
$ mpiexec -n 4 python demo.py
Hello World (from process 1)
Hello World (from process 2)
```

(continues on next page)

(continued from previous page)

```
Hello World (from process 3)
Hello World (from process 0)
```

The mpi4py package includes all kinds of mechanisms to share data between processes, synchronize, etc. It's a different flavor of parallelism than, say, threads or multiprocessing, but easy to get used to.

Check out the mpi4py web site for more information and a great tutorial.

5.5.2 Building against Parallel HDF5

HDF5 must be built with at least the following options:

```
$./configure --enable-parallel --enable-shared
```

Note that --enable-shared is required.

Often, a "parallel" version of HDF5 will be available through your package manager. You can check to see what build options were used by using the program h5cc:

```
$ h5cc -showconfig
```

Once you've got a Parallel-enabled build of HDF5, h5py has to be compiled in "MPI mode". Set your default compiler to the mpicc wrapper and build h5py with the HDF5_MPI environment variable:

```
$ export CC=mpicc
$ export HDF5_MPI="ON"
$ export HDF5_DIR="/path/to/parallel/hdf5"  # If this isn't found by default
$ pip install .
```

5.5.3 Using Parallel HDF5 from h5py

The parallel features of HDF5 are mostly transparent. To open a file shared across multiple processes, use the mpio file driver. Here's an example program which opens a file, creates a single dataset and fills it with the process ID:

```
from mpi4py import MPI
import h5py

rank = MPI.COMM_WORLD.rank  # The process ID (integer 0-3 for 4-process run)

f = h5py.File('parallel_test.hdf5', 'w', driver='mpio', comm=MPI.COMM_WORLD)

dset = f.create_dataset('test', (4,), dtype='i')
dset[rank] = rank

f.close()
```

Run the program:

```
$ mpiexec -n 4 python demo2.py
```

Looking at the file with h5dump:

```
$ h5dump parallel_test.hdf5
HDF5 "parallel_test.hdf5" {
GROUP "/" {
   DATASET "test" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 4 ) / ( 4 ) }
      DATA {
      (0): 0, 1, 2, 3
      }
   }
}
}
```

5.5.4 Collective versus independent operations

MPI-based programs work by launching many instances of the Python interpreter, each of which runs your script. There are certain requirements imposed on what each process can do. Certain operations in HDF5, for example, anything which modifies the file metadata, must be performed by all processes. Other operations, for example, writing data to a dataset, can be performed by some processes and not others.

These two classes are called *collective* and *independent* operations. Anything which modifies the *structure* or metadata of a file must be done collectively. For example, when creating a group, each process must participate:

```
>>> grp = f.create_group('x')  # right

>>> if rank == 1:
...     grp = f.create_group('x')   # wrong; all processes must do this
```

On the other hand, writing data to a dataset can be done independently:

```
>>> if rank > 2:
...     dset[rank] = 42   # this is fine
```

5.5.5 MPI atomic mode

HDF5 supports the MPI "atomic" file access mode, which trades speed for more stringent consistency requirements. Once you've opened a file with the `mpio` driver, you can place it in atomic mode using the settable `atomic` property:

```
>>> f = h5py.File('parallel_test.hdf5', 'w', driver='mpio', comm=MPI.COMM_WORLD)
>>> f.atomic = True
```

5.5.6 More information

Parallel HDF5 is a new feature in h5py. If you have any questions, feel free to ask on the mailing list (h5py at google groups). We welcome bug reports, enhancements and general inquiries.

5.6 Single Writer Multiple Reader (SWMR)

Starting with version 2.5.0, h5py includes support for the HDF5 SWMR features.

5.6.1 What is SWMR?

The SWMR features allow simple concurrent reading of a HDF5 file while it is being written from another process. Prior to this feature addition it was not possible to do this as the file data and meta-data would not be synchronised and attempts to read a file which was open for writing would fail or result in garbage data.

A file which is being written to in SWMR mode is guaranteed to always be in a valid (non-corrupt) state for reading. This has the added benefit of leaving a file in a valid state even if the writing application crashes before closing the file properly.

This feature has been implemented to work with independent writer and reader processes. No synchronisation is required between processes and it is up to the user to implement either a file polling mechanism, inotify or any other IPC mechanism to notify when data has been written.

The SWMR functionality requires use of the latest HDF5 file format: v110. In practice this implies using at least HDF5 1.10 (this can be checked via `h5py.version.info`) and setting the libver bounding to "latest" when opening or creating the file.

Warning: New v110 format files are *not* compatible with v18 format. So, files written in SWMR mode with libver='latest' cannot be opened with older versions of the HDF5 library (basically any version older than the SWMR feature).

The HDF Group has documented the SWMR features in details on the website: Single-Writer/Multiple-Reader (SWMR) Documentation. This is highly recommended reading for anyone intending to use the SWMR feature even through h5py. For production systems in particular pay attention to the file system requirements regarding POSIX I/O semantics.

5.6.2 Using the SWMR feature from h5py

The following basic steps are typically required by writer and reader processes:

- Writer process creates the target file and all groups, datasets and attributes.
- Writer process switches file into SWMR mode.
- Reader process can open the file with swmr=True.
- Writer writes and/or appends data to existing datasets (new groups and datasets *cannot* be created when in SWMR mode).
- Writer regularly flushes the target dataset to make it visible to reader processes.
- Reader refreshes target dataset before reading new meta-data and/or main data.
- Writer eventually completes and close the file as normal.
- Reader can finish and close file as normal whenever it is convenient.

The following snippet demonstrate a SWMR writer appending to a single dataset:

```
f = h5py.File("swmr.h5", 'w', libver='latest')
arr = np.array([1,2,3,4])
dset = f.create_dataset("data", chunks=(2,), maxshape=(None,), data=arr)
f.swmr_mode = True
# Now it is safe for the reader to open the swmr.h5 file
for i in range(5):
    new_shape = ((i+1) * len(arr), )
    dset.resize( new_shape )
    dset[i*len(arr):] = arr
    dset.flush()
    # Notify the reader process that new data has been written
```

The following snippet demonstrate how to monitor a dataset as a SWMR reader:

```
f = h5py.File("swmr.h5", 'r', libver='latest', swmr=True)
dset = f["data"]
while True:
    dset.id.refresh()
    shape = dset.shape
    print( shape )
```

5.6.3 Examples

In addition to the above example snippets, a few more complete examples can be found in the examples folder. These examples are described in the following sections.

Dataset monitor with inotify

The inotify example demonstrates how to use SWMR in a reading application which monitors live progress as a dataset is being written by another process. This example uses the the linux inotify (pyinotify python bindings) to receive a signal each time the target file has been updated.

```
"""
    Demonstrate the use of h5py in SWMR mode to monitor the growth of a dataset
    on notification of file modifications.

    This demo uses pyinotify as a wrapper of Linux inotify.
    https://pypi.python.org/pypi/pyinotify

    Usage:
            swmr_inotify_example.py [FILENAME [DATASETNAME]]

              FILENAME:    name of file to monitor. Default: swmr.h5
              DATASETNAME: name of dataset to monitor in DATAFILE. Default: data

    This script will open the file in SWMR mode and monitor the shape of the
    dataset on every write event (from inotify). If another application is
    concurrently writing data to the file, the writer must have have switched
    the file into SWMR mode before this script can open the file.
"""
```

(continues on next page)

(continued from previous page)

```
import asyncore
import pyinotify
import sys
import h5py
import logging

#assert h5py.version.hdf5_version_tuple >= (1,9,178), "SWMR requires HDF5 version >= 1.9.
↪178"

class EventHandler(pyinotify.ProcessEvent):

    def monitor_dataset(self, filename, datasetname):
        logging.info("Opening file %s", filename)
        self.f = h5py.File(filename, 'r', libver='latest', swmr=True)
        logging.debug("Looking up dataset %s"%datasetname)
        self.dset = self.f[datasetname]

        self.get_dset_shape()

    def get_dset_shape(self):
        logging.debug("Refreshing dataset")
        self.dset.refresh()

        logging.debug("Getting shape")
        shape = self.dset.shape
        logging.info("Read data shape: %s"%str(shape))
        return shape

    def read_dataset(self, latest):
        logging.info("Reading out dataset [%d]"%latest)
        self.dset[latest:]

    def process_IN_MODIFY(self, event):
        logging.debug("File modified!")
        shape = self.get_dset_shape()
        self.read_dataset(shape[0])

    def process_IN_CLOSE_WRITE(self, event):
        logging.info("File writer closed file")
        self.get_dset_shape()
        logging.debug("Good bye!")
        sys.exit(0)


if __name__ == "__main__":
    logging.basicConfig(format='%(asctime)s  %(levelname)s\t%(message)s',level=logging.
↪INFO)

    file_name = "swmr.h5"
    if len(sys.argv) > 1:
        file_name = sys.argv[1]
    dataset_name = "data"
```

(continues on next page)

(continued from previous page)

```
    if len(sys.argv) > 2:
        dataset_name = sys.argv[2]


    wm = pyinotify.WatchManager()  # Watch Manager
    mask = pyinotify.IN_MODIFY | pyinotify.IN_CLOSE_WRITE
    evh = EventHandler()
    evh.monitor_dataset( file_name, dataset_name )

    notifier = pyinotify.AsyncNotifier(wm, evh)
    wdd = wm.add_watch(file_name, mask, rec=False)

    # Sit in this loop() until the file writer closes the file
    # or the user hits ctrl-c
    asyncore.loop()
```

Multiprocess concurrent write and read

The SWMR multiprocess example starts two concurrent child processes: a writer and a reader. The writer process first creates the target file and dataset. Then it switches the file into SWMR mode and the reader process is notified (with a multiprocessing.Event) that it is safe to open the file for reading.

The writer process then continue to append chunks to the dataset. After each write it notifies the reader that new data has been written. Whether the new data is visible in the file at this point is subject to OS and file system latencies.

The reader first waits for the initial "SWMR mode" notification from the writer, upon which it goes into a loop where it waits for further notifications from the writer. The reader may drop some notifications, but for each one received it will refresh the dataset and read the dimensions. After a time-out it will drop out of the loop and exit.

```
"""
    Demonstrate the use of h5py in SWMR mode to write to a dataset (appending)
    from one process while monitoring the growing dataset from another process.

    Usage:
            swmr_multiprocess.py [FILENAME [DATASETNAME]]

              FILENAME:    name of file to monitor. Default: swmrmp.h5
              DATASETNAME: name of dataset to monitor in DATAFILE. Default: data

    This script will start up two processes: a writer and a reader. The writer
    will open/create the file (FILENAME) in SWMR mode, create a dataset and start
    appending data to it. After each append the dataset is flushed and an event
    sent to the reader process. Meanwhile the reader process will wait for events
    from the writer and when triggered it will refresh the dataset and read the
    current shape of it.
"""

import sys
import h5py
import numpy as np
import logging
from multiprocessing import Process, Event
```

(continues on next page)

(continued from previous page)

```
class SwmrReader(Process):
    def __init__(self, event, fname, dsetname, timeout = 2.0):
        super().__init__()
        self._event = event
        self._fname = fname
        self._dsetname = dsetname
        self._timeout = timeout

    def run(self):
        self.log = logging.getLogger('reader')
        self.log.info("Waiting for initial event")
        assert self._event.wait( self._timeout )
        self._event.clear()

        self.log.info("Opening file %s", self._fname)
        f = h5py.File(self._fname, 'r', libver='latest', swmr=True)
        assert f.swmr_mode
        dset = f[self._dsetname]
        try:
            # monitor and read loop
            while self._event.wait( self._timeout ):
                self._event.clear()
                self.log.debug("Refreshing dataset")
                dset.refresh()

                shape = dset.shape
                self.log.info("Read dset shape: %s"%str(shape))
        finally:
            f.close()

class SwmrWriter(Process):
    def __init__(self, event, fname, dsetname):
        super().__init__()
        self._event = event
        self._fname = fname
        self._dsetname = dsetname

    def run(self):
        self.log = logging.getLogger('writer')
        self.log.info("Creating file %s", self._fname)
        f = h5py.File(self._fname, 'w', libver='latest')
        try:
            arr = np.array([1,2,3,4])
            dset = f.create_dataset(self._dsetname, chunks=(2,), maxshape=(None,),␣
↪data=arr)
            assert not f.swmr_mode

            self.log.info("SWMR mode")
            f.swmr_mode = True
            assert f.swmr_mode
            self.log.debug("Sending initial event")
```

(continues on next page)

(continued from previous page)

```
            self._event.set()

            # Write loop
            for i in range(5):
                new_shape = ((i+1) * len(arr), )
                self.log.info("Resizing dset shape: %s"%str(new_shape))
                dset.resize( new_shape )
                self.log.debug("Writing data")
                dset[i*len(arr):] = arr
                #dset.write_direct( arr, np.s_[:], np.s_[i*len(arr):] )
                self.log.debug("Flushing data")
                dset.flush()
                self.log.info("Sending event")
                self._event.set()
        finally:
            f.close()


if __name__ == "__main__":
    logging.basicConfig(format='%(levelname)10s  %(asctime)s  %(name)10s  %(message)s',
↪level=logging.INFO)
    fname = 'swmrmp.h5'
    dsetname = 'data'
    if len(sys.argv) > 1:
        fname = sys.argv[1]
    if len(sys.argv) > 2:
        dsetname = sys.argv[2]

    event = Event()
    reader = SwmrReader(event, fname, dsetname)
    writer = SwmrWriter(event, fname, dsetname)

    logging.info("Starting reader")
    reader.start()
    logging.info("Starting reader")
    writer.start()

    logging.info("Waiting for writer to finish")
    writer.join()
    logging.info("Waiting for reader to finish")
    reader.join()
```

The example output below (from a virtual Ubuntu machine) illustrate some latency between the writer and reader:

```
python examples/swmr_multiprocess.py
  INFO  2015-02-26 18:05:03,195        root  Starting reader
  INFO  2015-02-26 18:05:03,196        root  Starting reader
  INFO  2015-02-26 18:05:03,197      reader  Waiting for initial event
  INFO  2015-02-26 18:05:03,197        root  Waiting for writer to finish
  INFO  2015-02-26 18:05:03,198      writer  Creating file swmrmp.h5
  INFO  2015-02-26 18:05:03,203      writer  SWMR mode
  INFO  2015-02-26 18:05:03,205      reader  Opening file swmrmp.h5
```

(continues on next page)

(continued from previous page)

```
INFO  2015-02-26 18:05:03,210      writer  Resizing dset shape: (4,)
INFO  2015-02-26 18:05:03,212      writer  Sending event
INFO  2015-02-26 18:05:03,213      reader  Read dset shape: (4,)
INFO  2015-02-26 18:05:03,214      writer  Resizing dset shape: (8,)
INFO  2015-02-26 18:05:03,214      writer  Sending event
INFO  2015-02-26 18:05:03,215      writer  Resizing dset shape: (12,)
INFO  2015-02-26 18:05:03,215      writer  Sending event
INFO  2015-02-26 18:05:03,215      writer  Resizing dset shape: (16,)
INFO  2015-02-26 18:05:03,215      reader  Read dset shape: (12,)
INFO  2015-02-26 18:05:03,216      writer  Sending event
INFO  2015-02-26 18:05:03,216      writer  Resizing dset shape: (20,)
INFO  2015-02-26 18:05:03,216      reader  Read dset shape: (16,)
INFO  2015-02-26 18:05:03,217      writer  Sending event
INFO  2015-02-26 18:05:03,217      reader  Read dset shape: (20,)
INFO  2015-02-26 18:05:03,218      reader  Read dset shape: (20,)
INFO  2015-02-26 18:05:03,219        root  Waiting for reader to finish
```

5.7 Virtual Datasets (VDS)

Starting with version 2.9, h5py includes high-level support for HDF5 'virtual datasets'. The VDS feature is available in version 1.10 of the HDF5 library; h5py must be built with a new enough version of HDF5 to create or read virtual datasets.

5.7.1 What are virtual datasets?

Virtual datasets allow a number of real datasets to be mapped together into a single, sliceable dataset via an interface layer. The mapping can be made ahead of time, before the parent files are written, and is transparent to the parent dataset characteristics (SWMR, chunking, compression etc…). The datasets can be meshed in arbitrary combinations, and even the data type converted.

Once a virtual dataset has been created, it can be read just like any other HDF5 dataset.

> **Warning:** Virtual dataset files cannot be opened with versions of the hdf5 library older than 1.10.

The HDF Group has documented the VDS features in detail on the website: Virtual Datasets (VDS) Documentation.

5.7.2 Creating virtual datasets in h5py

To make a virtual dataset using h5py, you need to:

1. Create a *VirtualLayout* object representing the dimensions and data type of the virtual dataset.
2. Create a number of *VirtualSource* objects, representing the datasets the array will be built from. These objects can be created either from an h5py *Dataset*, or from a filename, dataset name and shape. This can be done even before the source file exists.
3. Map slices from the sources into the layout.
4. Convert the *VirtualLayout* object into a virtual dataset in an HDF5 file.

The following snippet creates a virtual dataset to stack together four 1D datasets from separate files into a 2D dataset:

```
layout = h5py.VirtualLayout(shape=(4, 100), dtype='i4')

for n in range(1, 5):
    filename = "{}.h5".format(n)
    vsource = h5py.VirtualSource(filename, 'data', shape=(100,))
    layout[n - 1] = vsource

# Add virtual dataset to output file
with h5py.File("VDS.h5", 'w', libver='latest') as f:
    f.create_virtual_dataset('data', layout, fillvalue=-5)
```

This is an extract from the `vds_simple.py` example in the examples folder.

Note: Slices up to `h5py.h5s.UNLIMITED` can be used to create an unlimited selection along a single axis. Resizing the source data along this axis will cause the virtual dataset to grow. E.g.:

```
layout[n - 1, :UNLIMITED] = vsource[:UNLIMITED]
```

A normal slice with no defined end point (`[:]`) is fixed based on the shape when you define it.

New in version 3.0.

5.7.3 Examples

In addition to the above example snippet, a few more complete examples can be found in the examples folder:

- vds_simple.py is a self-contained, runnable example which creates four source files, and then maps them into a virtual dataset as shown above.
- dataset_concatenation.py illustrates virtually stacking datasets together along a new axis.
- A number of examples are based on the sample use cases presented in the virtual datasets RFC:
  - excalibur_detector_modules.py
  - dual_pco_edge.py
  - eiger_use_case.py
  - percival_use_case.py

5.7.4 Reference

class h5py.**VirtualLayout**(*shape*, *dtype*, *maxshape=None*)

Object for building a virtual dataset.

Instantiate this class to define a virtual dataset, assign *VirtualSource* objects to slices of it, and then pass it to *Group.create_virtual_dataset()* to add the virtual dataset to a file.

This class does not allow access to the data; the virtual dataset must be created in a file before it can be used.

Parameters

- **shape** (*tuple*) – The full shape of the virtual dataset.

- **dtype** – Numpy dtype or string.
- **maxshape** (*tuple*) – The virtual dataset is resizable up to this shape. Use None for axes you want to be unlimited.

class h5py.**VirtualSource**(*path_or_dataset*, *name=None*, *shape=None*, *dtype=None*, *maxshape=None*)

Source definition for virtual data sets.

Instantiate this class to represent an entire source dataset, and then slice it to indicate which regions should be used in the virtual dataset.

When *creating a virtual dataset*, paths to sources present in the same file are changed to a ".", referring to the current file (see H5Pset_virtual). This will keep such sources valid in case the file is renamed.

Parameters

- **path_or_dataset** – The path to a file, or a *Dataset* object. If a dataset is given, no other parameters are allowed, as the relevant values are taken from the dataset instead.
- **name** (*str*) – The name of the source dataset within the file.
- **shape** (*tuple*) – The full shape of the source dataset.
- **dtype** – Numpy dtype or string.
- **maxshape** (*tuple*) – The source dataset is resizable up to this shape. Use None for axes you want to be unlimited.

5.8 Tools and Related Projects

There are a number of projects which build upon h5py, or who build upon HDF5, which will likely be of interest to users of h5py. This page is non-exhaustive, but if you think there should be a project added, feel free to create an issue or pull request at https://github.com/h5py/h5py/.

PyTables is the most significant related project, providing a higher level wrapper around HDF5 then h5py, and optimised to fully take advantage of some of HDF5's features. h5py provides a comparison between the two projects (see *What's the difference between h5py and PyTables?*), as does the PyTables project.

See also:

HDF Group's list of HDF5 tools

- *IPython*
- *Exploring and Visualising HDF5 files*
- *Additional Filters*
- *Libraries extending h5py*

5.8.1 IPython

H5py ships with a custom ipython completer, which provides object introspection and tab completion for h5py objects in an ipython session. For example, if a file contains 3 groups, "foo", "bar", and "baz":

```
In [4]: f['b<TAB>
bar   baz

In [4]: f['f<TAB>
# Completes to:
In [4]: f['foo'

In [4]: f['foo'].<TAB>
f['foo'].attrs            f['foo'].items            f['foo'].ref
f['foo'].copy             f['foo'].iteritems        f['foo'].require_dataset
f['foo'].create_dataset   f['foo'].iterkeys         f['foo'].require_group
f['foo'].create_group     f['foo'].itervalues       f['foo'].values
f['foo'].file             f['foo'].keys             f['foo'].visit
f['foo'].get              f['foo'].name             f['foo'].visititems
f['foo'].id               f['foo'].parent
```

The easiest way to enable the custom completer is to do the following in an IPython session:

```
In [1]: import h5py

In [2]: h5py.enable_ipython_completer()
```

The completer can be enabled for every session by adding "h5py.ipy_completer" to the list of extensions in your ipython config file, for example `~/.config/ipython/profile_default/ipython_config.py` (if this file does not exist, you can create it by invoking *ipython profile create*):

```
c = get_config()
c.InteractiveShellApp.extensions = ['h5py.ipy_completer']
```

5.8.2 Exploring and Visualising HDF5 files

h5py does not contain a tool for exploring or visualising HDF5 files, but tools that can display the structure of h5py include:

- HDFView is a visual tool for browsing and editing HDF5 files.
- ViTables is a GUI for browsing and editing files in both PyTables and HDF5 formats, and is built on top of PyTables.
- h5glance shows the structure of HDF5 files in IPython & Jupyter, as well as at the command line.

See also:

The PaNOSC project's list of HDF5 & NeXus viewers

5.8.3 Additional Filters

Some projects providing additional HDF5 filter with integration into h5py include:

- hdf5plugin: this provides several plugins (currently Blosc, Blosc2, BitShuffle, BZip2, FciDecomp, LZ4, SZ, SZ3, Zfp, ZStd), and newer plugins should look to supporting h5py via inclusion into hdf5plugin.

5.8.4 Libraries extending h5py

These libraries offer additional general functionality on top of h5py:

- Versioned HDF5 offers a versioned abstraction on top of h5py. It provides a wrapper around the h5py API that allows storing different versions of groups and datasets within an HDF5 file.
- h5preserve lets you define how to save and load instances of a given class in HDF5 files, by writing dumper and loader functions. These functions can also have multiple versions.
- Hickle provides an API like `pickle` to dump & load arbitrary Python objects in HDF5 files.
- h5pickle wraps h5py to allow pickling objects such as *`File`* or *`Dataset`*. This relies on the file being available at the same path when unpickling.
- b2h5py provides h5py with transparent, automatic optimized reading of n-dimensional slices of Blosc2-compressed datasets, using direct chunk access and 2-level partitioning.

CHAPTER

SIX

META-INFO ABOUT THE H5PY PROJECT

6.1 "What's new" documents

These document the changes between minor (or major) versions of h5py.

6.1.1 What's new in h5py 3.10

New features

- h5py now has pre-built packages for Python 3.12.
- Pre-built packages on Linux & Windows now bundle HDF5 version 1.14.2. Mac packages still contain HDF5 1.12.2 for now. You can still *build h5py from source* against a wider range of HDF5 versions.
- The read-only S3 file driver ('ros3') now accepts an AWS session token as part of the credentials (PR 2301). Pass `session_token` when opening a *`File`* (along with the other S3 parameters). This requires HDF5 1.14.2 or later, with the ROS3 feature built.

Deprecations & removals

- Support for the HDF5 1.8 series was dropped, along with early 1.10 releases. The minimum required HDF5 version is now 1.10.4.

Exposing HDF5 functions

- `H5Pget_fapl_ros3_token` & `H5Pset_fapl_ros3_token`

Bug fixes

- Various nasty bugs when using nested compound and vlen data types have been fixed (PR 2134).
- Fixed an `OverflowError` in some cases when registering a filter with `h5z.register_filter()`, especially on 32-bit architectures (PR 2318).
- Sequential slicing/indexing operations on a *`VirtualSource`* object (e.g. `source[:10][::2]`) now raise an error, rather than giving incorrect results (PR 2280).

Building h5py

- h5py now uses HDF5's 1.10 compatibility mode at compile time rather than the 1.8 compatibility mode (PR 2320). This is normally transparent even if you're building h5py from source.

6.1.2 What's new in h5py 3.9

This version of h5py requires Python 3.8 or above.

New features

- New `out` argument to `read_direct_chunk()` to allow passing the output buffer (PR 2232).
- The objects from *`Dataset.asstr()`* and *`Dataset.astype()`* now implement the `__array__()` method (PR 2269). This speeds up access for functions that support it, such as `np.asarray()`.
- Validate key types when creating groups and attributes, giving better error messages when invalid types are used (PR 2266).

Deprecations & removals

- Using *`Dataset.astype()`* as a context manager has been removed, after being deprecated in h5py 3.6. Read data by slicing the returned object instead: `dset.astype('f4')[:]`.

Exposing HDF5 functions

- `H5Pget_elink_acc_flags` & `H5Pset_elink_acc_flags` as `h5py.h5p.PropLAID.get_elink_acc_flags()` & `h5py.h5p.PropLAID.set_elink_acc_flags()`: access the external link file access traversal flags in a link access property list (PR 2244).
- `H5Zregister` as `h5py.h5z.register_filter()`: register an HDF5 filter (PR 2229).

Bug fixes

- `Group.__contains__` and `Group.get` now use the default link access property list systematically (PR 2244).
- Removed various calls to the deprecated `numpy.product` function (PR 2242 & PR 2273).
- Fix the IPython tab-completion integration in IPython 8.12 (:pr:2256`).
- Replacing attributes with *`AttributeManager.create()`* now deletes the old attributes before creating the new one, rather than using a temporary name and renaming the new attribute (PR 2274). This should avoid some confusing bugs affecting attributes. However, failures creating an attribute are less likely to leave an existing attribute of the same name in place. To change an attribute value without changing its shape or dtype, use *`modify()`* instead.

Building h5py

- When building with *Parallel HDF5* support, the version of mpi4py used on various Python versions is increased to 3.1.1, fixing building with a newer setuptools (PR 2225).
- Some fixes towards compatibility with the upcoming Cython 3 (PR 2247).

6.1.3 What's new in h5py 3.8

New features

- h5py now has pre-built packages for Python 3.11.
- h5py is compatible with HDF5 1.14 (PR 2187). Pre-built packages on PyPI still include HDF5 1.12 for now.
- *Fancy indexing* now accepts tuples, or any other sequence type, rather than only lists and NumPy arrays. This also includes `range` objects, but this will normally be less efficient than the equivalent slice.
- New property *`Dataset.is_scale`* for checking if the dataset is a dimension scale (PR 2168).
- *`Group.require_dataset()`* now validates `maxshape` for resizable datasets (PR 2116).
- *`File`* now has a `meta_block_size` argument and property. This influences how the space for metadata, including the initial header, is allocated.
- Chunk cache can be configured per individual HDF5 dataset (PR 2127). Use *`Group.create_dataset()`* for new datasets or *`Group.require_dataset()`* for already existing datasets. Any combination of the `rdcc_nbytes`, `rdcc_w0`, and `rdcc_nslots` arguments is allowed. The file defaults apply to those omitted.
- HDF5 file names for ros3 driver can now also be `s3://` resource locations (PR 2140). h5py will translate them into AWS path-style URLs for use by the driver.
- When using the ros3 driver, AWS authentication will be activated only if all three driver arguments are provided. Previously AWS authentication was active if any one of the arguments was set causing an error from the HDF5 library.
- *`Dataset.fields()`* now implements the `__array__()` method (PR 2151). This speeds up accessing fields with functions that expect this, like `np.asarray()`.
- Low-level `h5py.h5d.DatasetID.chunk_iter()` method that invokes a user-supplied callable object on every written chunk of one dataset (PR 2202). It provides much better performance when iterating over a large number of chunks.

Exposing HDF5 functions

- `H5Dchunk_iter` as `h5py.h5d.DatasetID.chunk_iter()`.
- H5Pset_meta_block_size and H5Pget_meta_block_size (PR 2106).

Bug fixes

- Fixed getting the default fill value (an empty string) for variable-length string data (PR 2132).
- Complex float16 data could cause a `TypeError` when trying to coerce to the currently unavailable numpy.dtype('c4'). Now a compound type is used instead (PR 2157).
- h5py 3.7 contained a performance regression when using a boolean mask array to index a 1D dataset, which is now fixed (PR 2193).

Building h5py

- Parallel HDF5 can be built with Microsoft MS-MPI (PR 2147). See *Building against Parallel HDF5* for details.
- Some 'incompatible function pointer type' compile time warnings were fixed (PR 2142).
- Fix for finding HDF5 DLL in mingw (PR 2105).

6.1.4 What's new in h5py 3.7

New features

- Both Apple Silicon (arm64) and Intel (x86_64) Mac wheels are now provided (PR 2065). Apple Silicon wheels are not automatically tested, however, as we're not aware of any CI offerings that do this.
- Provide the ability to use the Direct Virtual File Driver (VFD) from HDF5 (Linux only). If the Direct VFD driver is present at the time of compilation, users can use the Direct VFD by passing the keyword argument `driver="direct"` to the `h5py.File` constructor.

  To use the Direct VFD, HDF5 and h5py must have both been compiled with this enabled. Currently, pre-built h5py wheels on PyPI do not include the Direct VFD. Other packages such as the conda package on conda-forge might include it. Alternatively, you can *build h5py from source* against an HDF5 build with the direct driver enabled.
- The *File* constructor contains two new parameters `alignment_threshold`, and `alignment_interval` controlling the data alignment within the HDF5 file (PR 2040).
- *create_dataset()* and *require_dataset()* now accept parameters `efile_prefix` and `virtual_prefix` to set a filesystem path prefix to use to find files for external datasets and for virtual dataset sources (PR 2092). These only affect the current access; the prefix is not stored in the file.
- h5py wheels on PyPI now bundle HDF5 version 1.12.2 (PR 2099).
- h5py Mac wheels on PyPI now bundle zlib version 1.2.12 (PR 2082).
- Pre-built wheels are now available for Python 3.10 on Linux ARM64 (PR 2094).

Bug fixes

- Fix a deadlock which was possible when the same dataset was accessed from multiple threads (GH2064).
- New attributes are created directly, instead of via a temporary attribute with subsequent renaming. This fixes overwriting attributes with `track_order=True`.
- Fix for building with mpi4py on Python 3.10 (PR 2101).
- Fixed fancy indexing with a boolean array for a single dimension (PR 2079).

- Avoid returning uninitialised memory when reading from a chunked dataset with missing chunks and no fill value (PR 2076).
- Enable setting of fillvalue for datasets with variable length string dtype (PR 2044).
- Closing a file or calling `get_obj_ids()` no longer re-enables Python garbage collection if it was previously disabled (PR 2020).

Exposing HDF5 functions

- `H5Pset_efile_prefix` and `H5Pget_efile_prefix`

Building h5py

- Fix for building h5py on Cygwin (PR 2038).
- More helpful error message when `pkg-config` is unavailable (PR 2053).

6.1.5 What's new in h5py 3.6

New features

- Pre-built packages are now available for Python 3.10.

Deprecations

- Using *`Dataset.astype()`* as a context manager (`with dset.astype(t):`) is deprecated. Slice the object returned by astype instead (`data = dset.astype(t)[:10]`). This works from h5py 3.0 onwards.
- Getting the value of `h5py.get_config().default_file_mode` now issues a deprecation warning. This has been `'r'` by default from h5py 3.0, and cannot be changed since 3.3.

Building h5py

- h5py now requires the `oldest-supported-numpy` package at build time, instead of maintaining its own list of the oldest supported NumPy versions. The effect should be similar, but hopefully more reliable.

Development

- The custom `setup.py test` has been removed. tox should be used instead during development (see *Run the tests*), and `pytest --pyargs h5py` can be used to test h5py after installation.

6.1.6 What's new in h5py 3.5

New features

- Datasets are now created without timestamps by default, making it easier to create more consistent files. Pass `track_times=True` to *Group.create_dataset()* to add timestamps again.
- Added `locking` *File* argument to select HDF5 file locking behavior.
- Enable setting file space page size when creating new HDF5 files. A new named argument `fs_page_size` is added to the *File* class.
- Enable HDF5 page buffering, a low-level caching feature that may improve overall I/O performance in some cases. Three new named arguments are added to the *File* class: `page_buf_size`, `min_meta_keep`, and `min_raw_keep`.
- Get and reset HDF5 page buffering statistics. Available as the low-level API of the `FileID` class.
- The built-in `reversed()` function now works with various dictionary-like interfaces: *Group*, `GroupID`, *Group.keys()*, *Group.values()* and *Group.items()*.

Exposing HDF5 functions

- `H5Pset_file_locking` and `H5Pget_file_locking` (for HDF5 >= 1.12.1 or 1.10.x >= 1.10.7)
- `H5Freset_page_buffering_stats`
- `H5Fget_page_buffering_stats`
- `H5Pset_file_space_page_size`
- `H5Pget_file_space_page_size`
- `H5Pset_page_buffer_size`
- `H5Pget_page_buffer_size`

Breaking changes & deprecations

- Dataset timestamps are no longer written by default for new datasets. Pass `track_times=True` to *Group.create_dataset()* if you need them.
- The IPython completer code no longer tries to work with very old versions of IPython (before 1.0).

Bug fixes

- Fix a memory leak when reading data. This particularly affected code making many small reads.
- `dataset == array` now behaves the same way as `array == dataset`: the HDF5 dataset is read and NumPy makes a boolean array.
- The IPython completer code no longer imports the `readline` module.

6.1.7 What's new in h5py 3.4

New features

- The pre-built wheels now bundle HDF5 1.12.1 (PR 1945).
- `len()` now works on `dset.astype()`, `.asstr()` and `.fields()` wrappers (PR 1913).

Bug fixes

- Fix bug introduced in version 3.3 that did not allow the creation of files using the flag "a" for certain drivers (e.g. mpiio, core and stdio) (PR 1922).
- Dataset indexing will now use the optimized fast path, which was accidentally disabled in a previous version (PR 1944).
- Fix an error building with Cython 3.0 alpha 8 (`cpdef` inside functions) (PR 1923).

6.1.8 What's new in h5py 3.3

New features

- Compatibility with the upcoming HDF5 1.12.1 and possibly 1.14 (PR 1875).
- H5T_BITFIELD types will now be cast to their `numpy.uint` equivalent by default (GH1258). This means that no knowledge of mixed type compound dataset schemas is required to read these types, and can simply be read as follows:

```
arr = dset[:]
```

  Alternatively, 8-bit bitfields can still be cast to booleans explicitly:

```
arr = dset.astype(numpy.bool_)[:]
```

- Key types are validated when accessing groups, to give more helpful errors when a group is indexed like a dataset (PR 1856).
- A new *`Group.build_virtual_dataset()`* method acting as a context manager to assemble virtual datasets (PR 1905).
- If the source and target of a virtual dataset mapping have different numbers of points, an error should now be thrown when you make the mapping in the `VirtualLayout`, rather than later when writing this into the file. This should make it easier to find the source of such errors.

Deprecations

- Linux wheels are now manylinux2010 rather than manylinux1
- The `default_file_mode` config option is deprecated, and setting it to values other than 'r' (for read-only mode) is no longer allowed. Pass the mode when creating a *`File`* object instead of setting a global default.

Bug fixes

- Trying to open a file in append mode (`'a'`) should now give clearer error messages when the file exists but can't be opened (PR 1902).
- Protect `h5py.h5f.get_obj_ids()` against garbage collection invalidating HDF5 IDs while it is retrieving them (GH1852).
- Make file closing more robust, including when closing files while the interpreter is shutting down, by using lower-level code to close HDF5 IDs of objects inside the file (GH1495).

6.1.9 What's new in h5py 3.2

New features

- Added support to use the HDF5 ROS3 driver to access HDF5 files on S3 (PR 1755). This is not enabled in the pre-built packages on PyPI. To use it, ensure HDF5 is built with read-only S3 support enabled, and then *build h5py from source* using that HDF5 library.

Breaking changes & deprecations

- Python 3.7 is now the minimum supported version. It may still be possible to use this release with Python 3.6, but it isn't tested and wheels are not provided for Python 3.6.
- Setting the config option `default_file_mode` to values other than `'r'` is deprecated. Pass the desired mode when opening a *`File`* instead.

Exposing HDF5 functions

- `H5Pset_fapl_ros3` & `H5Pget_fapl_ros3` (where HDF5 is built with read-only S3 support).

Bug fixes

- `OSError` exceptions raised by h5py should now have a useful `.errno` attribute, where HDF5 provides this information. Subclasses such as `FileNotFoundError` should also be raised where appropriate (PR 1815).
- Fix reading data with a datatype of variable-length arrays of fixed length strings (GH1817).
- Fix *`Dataset.read_direct()`* and *`Dataset.write_direct()`* when the source and destination have different shapes (PR 1796).
- Fix selecting data using integer indices in *`Dataset.read_direct()`* and *`Dataset.write_direct()`* (PR 1818).
- Fix exception handling in *`Group.visititems()`* (GH1740).
- Issue a warning when `File(..., swmr=True)` is specified with any mode other than `'r'`, as the SWMR option is ignored in these cases (PR 1812).
- Fix NumPy 1.20 deprecation warnings concerning the use of None as shape, and the deprecated aliases np.float, np.int and np.bool (PR 1780).

3.2.1 bug fix release

- Fix *File.driver* when the read-only S3 driver is available (PR 1844).

6.1.10 What's new in h5py 3.1

Bug fixes

- Fix reading numeric data which is not in the native endianness, e.g. big-endian data on a little-endian system (GH1729).
- Fix using bytes as names for *Group.create_dataset()* and *Group.create_virtual_dataset()* (GH1732).
- Fix writing data as a list to a dataset with a sub-array data type (GH1735).

Building h5py

- Allow building against system lzf library by setting `H5PY_SYSTEM_LZF=1`. See *Custom installation*.

Development

- If pytest is missing pytest-mpi it will now fail with a clear error.
- Fix a test which was failing on big-endian systems.

6.1.11 What's new in h5py 3.0

New features

- The interface for storing & reading strings has changed - see *Strings in HDF5*. The new rules are hopefully more consistent, but may well require some changes in coding using h5py.
- Reading & writing data now releases the GIL, so another Python thread can continue while HDF5 accesses data. Where HDF5 can call back into Python, such as for data conversion, h5py re-acquires the GIL. However, HDF5 has its own global lock, so this won't speed up parallel data access using multithreading.
- Numpy datetime and timedelta arrays can now be stored and read as HDF5 opaque data (GH1339), though other tools will not understand them. See *Storing other types as opaque data* for more information.
- New *Dataset.iter_chunks()* method, to iterate over chunks within the given selection.
- Compatibility with HDF5 1.12.
- Methods which accept a shape tuple, e.g. to create a dataset, now also allow an integer for a 1D shape (PR 1340).
- Casting data to a specified type on reading (*Dataset.astype()*) can now be done without a with statement, like this:

```
data = dset.astype(np.int32)[:]
```

- A new *Dataset.fields()* method lets you read only selected fields from a dataset with a compound datatype.
- Reading data has less overhead, as selection has been implemented in Cython. Making many small reads from the same dataset can be as much as 10 times faster, but there are many factors that can affect performance.

- A new NumPy-style *Dataset.nbytes* attribute to get the size of the dataset's data in bytes. This differs from the *size* attribute, which gives the number of elements.
- The `external` argument of *Group.create_dataset()*, which specifies any external storage for the dataset, accepts more types (GH1260), as follows:
  - The top-level container may be any iterable, not only a list.
  - The names of external files may be not only `str` but also `bytes` or `os.PathLike` objects.
  - The offsets and sizes may be NumPy integers as well as Python integers.

  See also the deprecation related to the `external` argument.
- Support for setting file space strategy at file creation. Includes option to persist empty space tracking between sessions. See *File* for details.
- More efficient writing when assigning a scalar to a chunked dataset, when the number of elements to write is no more than the size of one chunk.
- Introduced support for the split *file driver* (PR 1468).
- Allow making virtual datasets which can grow as the source data is resized - see *Virtual Datasets (VDS)*.
- New *allow_unknown_filter* option to *Group.create_dataset()*. This should only be used if you will compress the data before writing it with the low-level `write_direct_chunk()` method.
- The low-level chunk query API provides information about dataset chunks in an HDF5 file: `get_num_chunks()`, `get_chunk_info()` and `get_chunk_info_by_coord()`.
- The low-level `h5py.h5f.FileID.get_vfd_handle()` method now works for any file driver that supports it, not only the sec2 driver.

Breaking changes & deprecations

- h5py now requires Python 3.6 or above; it is no longer compatible with Python 2.7.
- The default mode for opening files is now 'r' (read-only). See *Opening & creating files* for other possible modes if you need to write to a file.
- In previous versions, creating a dataset from a list of bytes objects would choose a fixed length string datatype to fit the biggest item. It will now use a variable length string datatype. To store fixed length strings, use a suitable dtype from *h5py.string_dtype()*.
- Variable-length UTF-8 strings in datasets are now read as `bytes` objects instead of `str` by default, for consistency with other kinds of strings. See *Strings in HDF5* for more details.
- When making a virtual dataset, a dtype must be specified in *VirtualLayout*. There is no longer a default dtype, as this was surprising in some cases.
- The `external` argument of `Group.create_dataset()` no longer accepts the following forms (GH1260):
  - a list containing *name*, [*offset*, [*size*]];
  - a list containing *name1*, *name2*, ...; and
  - a list containing tuples such as `(name,)` and `(name, offset)` that lack the offset or size.

  Furthermore, each *name–offset–size* triplet now must be a tuple rather than an arbitrary iterable. See also the new feature related to the `external` argument.
- The MPI mode no longer supports mpi4py 1.x.
- The deprecated `h5py.h5t.available_ftypes` dictionary was removed.

- The deprecated `Dataset.value` property was removed. Use `ds[()]` to read all data from any dataset.
- The deprecated functions `new_vlen`, `new_enum`, `get_vlen` and `get_enum` have been removed. See *Special types* for the newer APIs.
- Removed deprecated File.fid attribute. Use *`File.id`* instead.
- Remove the deprecated `h5py.highlevel` module. The high-level API is available directly in the `h5py` module.
- The third argument of `h5py._hl.selections.select()` is now an optional high-level *`Dataset`* object, rather than a `DatasetID`. This is not really a public API - it has to be imported through the private `_hl` module - but probably some people are using it anyway.

Exposing HDF5 functions

- H5Dget_num_chunks
- H5Dget_chunk_info
- H5Dget_chunk_info_by_coord
- H5Oget_info1
- H5Oget_info_by_name1
- H5Oget_info_by_idx1
- H5Ovisit1
- H5Ovisit_by_name1
- H5Pset_attr_phase_change
- H5Pset_fapl_split
- H5Pget_file_space_strategy
- H5Pset_file_space_strategy
- H5Sencode1
- H5Tget_create_plist

Bug fixes

- Fix segmentation fault when accessing vlen of strings (GH1336).
- Fix the storage of non-contiguous arrays, such as numpy slices, as HDF5 vlen data (GH1649).
- Fix pathologically slow reading/writing in certain conditions with integer indexing (GH492).
- Fix bug when *`Group.copy()`* source is a high-level object and destination is a Group (GH1005).
- Fix reading data for region references pointing to an empty selection.
- Unregister converter functions at exit, preventing segfaults on exit in some situations with threads (PR 1440).
- As HDF5 1.10.6 and later support UTF-8 paths on Windows, h5py built against HDF5 1.10.6 will use UTF-8 for file names, allowing all filenames.
- Fixed `h5py.h5d.DatasetID.get_storage_size()` to report storage size of zero bytes without raising an exception (GH1475).
- Attribute Managers (`obj.attrs`) can now work on HDF5 stored datatypes (GH1476).

- Remove broken inherited `ds.dims.values()` and `ds.dims.items()` methods. The dimensions interface behaves as a sequence, not a mapping (GH744).
- Fix creating attribute with `Empty` by converting its dtype to a numpy dtype object.
- Fix getting *`maxshape`* on empty/null datasets.
- The *`File.swmr_mode`* property is always available (GH1580).
- The *`File.mode`* property handles SWMR access modes in addition to plain RDONLY/RDWR modes
- Importing an MPI build of h5py no longer initialises MPI immediately, which will hopefully avoid various strange behaviours.
- Avoid launching a subprocess by using `platform.machine()` at import time. This could trigger a warning in MPI.
- Removed an equality comparison with an empty array, which will cause problems with future versions of numpy.
- Better error message if you try to use the mpio driver and h5py was not built with MPI support.
- Improved error messages when requesting chunked storage for an empty dataset.
- Data conversion functions should fail more gracefully if no memory is available.
- Fix some errors for internal functions that were raising "TypeError: expected bytes, str found" instead of the correct error.
- Use relative path for virtual data sources if the source dataset is in the same file as the virtual dataset.
- Generic exception types used in tests' assertRaise (exception types changed in new HDF5 version)
- Use `dtype=object` in tests with ragged arrays

Building h5py

- The `setup.py configure` command was removed. Configuration for the build can be specified with environment variables instead. See *Custom installation* for details.
- It is now possible to specify separate include and library directories for HDF5 via environment variables. See *Custom installation* for more details.
- The pkg-config name to use when looking up the HDF5 library can now be configured, this can assist with selecting the correct HDF5 library when using MPI. See *Custom installation* for more details.
- Using bare `char*` instead of `array.array` in h5d.read_direct_chunk since `array.array` is a private CPython C-API interface
- Define `NPY_NO_DEPRECATED_API` to silence a warning.
- Make the lzf filter build with HDF5 1.10 (GH1219).
- If HDF5 is not loaded, an additional message is displayed to check HDF5 installation
- Rely much more on the C-interface provided by Cython to call Python and NumPy.
- Removed an old workaround which tried to run Cython in a subprocess if cythonize() didn't work. This shouldn't be necessary for any recent version of setuptools.
- Migrate all Cython code base to Cython3 syntax
  - The only noticeable change is in exception raising from cython which use bytes
  - Massively use local imports everywhere as expected from Python3
  - Explicitly mark several Cython functions as non-binding

Development

- Unregistering converter functions on exit (PR 1440) should allow profiling and code coverage tools to work on Cython code.

6.1.12 What's new in h5py 2.10

New features

- HDF5 8-bit bitfield data can now be read either as uint8 or booleans (GH821). Pytables stores booleans as this type. For now, you must pick which type to use explicitly:

```
with dset.astype(numpy.uint8):   # or numpy.bool
    arr = dset[:]
```

- Numpy arrays of integers can now be used for fancy indexing, where previously a Python list was required (GH963).
- Fancy indexing now allows an empty list or array (GH1174).
- IPython can now tab-complete names in h5py groups and attributes without any special user action (GH1228). This simple completion only matches the first level of keys in a group, not subkeys. You can still call `h5py.enable_ipython_completion()` for more complete results.
- The `libver` parameter for `File` now accepts `'v108'` and `'v110'` to specify compatibility with HDF5 1.8 or 1.10 (GH1155). See *Version bounding* for details.
- New functions and constants for getting and identifying *special data types* - `string_dtype()`, `vlen_dtype()`, `enum_dtype()`, `ref_dtype` and `regionref_dtype` replace `special_dtype()`. For identifying string, vlen and enum dtypes, `check_string_dtype()`, `check_vlen_dtype()` and `check_enum_dtype()` replace `check_dtype()` (GH1132).
- A new method *`make_scale()`* to conveniently make a dataset into a *dimension scale* (GH830, GH1212).
- A new method `AttributeManager.get_id()` to get a low-level `AttrID` object referring to an attribute (GH1278).
- Several examples were updated to run on Python 3 (GH1149).

Deprecations

- The default behaviour of `h5py.File` with no specified mode is deprecated (GH1143). It currently tries to create a file or open it for read/write access, silently falling back to read-only depending on permissions. From h5py 3.0, the default will be read-only.

  Ideally, code should pass an explicit mode each time a file is opened:

```
h5py.File("example.h5", "r")
```

  The possible modes are described in *Opening & creating files*. If you want to suppress the deprecation warnings from code you can't modify, you can either:

  - set `h5.get_config().default_file_mode = 'r'` (or another available mode)
  - or set the environment variable `H5PY_DEFAULT_READONLY` to any non-empty string, to adopt the future default.

- This is expected to be the last h5py release to support Python 2.7 and 3.4.

Exposing HDF5 functions

- `H5Zunregister` exposed as `h5z.unregister_filter()` (GH746, GH1224).
- The new module `h5py.h5pl` module exposes various `H5PL` functions to inspect and modify the search path for plugins (GH1166, GH1256).
- `H5Dread_chunk` exposed as `h5d.read_direct_chunk()` (GH1190).

Bugfixes

- Fix crash with empty variable-length data (GH1248, GH1253).
- Fixed random selection of data type when reading 64-bit floats on Windows where Python uses random dictionary order (GH1051, GH1134).
- Pickling h5py objects now fails explicitly. It previously failed on unpickling, and we can't reliably serialise and restore handles to HDF5 objects anyway (GH531, GH1194). If you need to use these objects in other processes, you could explicitly serialise the filename and the name of the object inside the file. Or consider h5pickle, which does the same implicitly.
- Creating a dataset with external storage can no longer mutate the `external` list parameter passed in (GH1205). It also has improved error messages (GH1204).
- Certain deprecation warnings will now show the relevant line of code which uses the deprecated feature (GH1146).
- Skipped a failing test for complex floating point numbers on 32-bit x86 systems (GH1235).
- Disabled the longdouble type on the `ppc64le` architecture, as it was causing segfaults with more commonly used float types (GH1243).
- Documented that nested compound types are not currently supported (GH1236).
- Fixed attribute `create` method to be consistent with `__setattr__` (GH1265).

Building h5py

- The version of HDF5 can now be automatically detected on Windows (GH1123).
- Fixed autodetecting the version from libhdf5 in default locations on Windows and Mac (GH1240).
- Fail to build if it can't detect version from libhdf5, rather than assuming 1.8.4 as a default (GH1241).
- Building h5py from source on Unix platforms now requires either `pkg-config` or an explicitly specified path to HDF5 (GH1231). Previously it had a hardcoded default path, but when this was wrong, the failures were unnecessarily confusing.
- The Cython 'language level' is now explicitly set to 2, to prepare h5py for changing defaults in Cython (GH1171).
- Avoid using `setup_requires` when pip calls `setup.py egg_info` (GH1259).

Development

- h5py's tests are now run by pytest (GH1003), and coverage reports are automatically generated on Codecov.

6.1.13 What's new in h5py 2.9

New features

- A convenient high-level API for creating virtual datasets, HDF5 objects which act as a view over one or more real datasets (GH1060, GH1126). See *Virtual Datasets (VDS)* for details.
- `File` can now be constructed with a Python file-like object, making it easy to create an HDF5 file in memory using `io.BytesIO` (GH1061, GH1105, GH1116). See *Python file-like objects* for details.
- `File` now accepts parameters to control the chunk cache (GH1008). See *Chunk cache* for details.
- New options to record the order of insertion for attributes and group entries. Iterating over these collections now follows insertion order if it was recorded, or name order if not (GH1098).
- A new method `Group.create_dataset_like()` to create a new dataset with similar properties to an existing one (GH1085).
- Datasets can now be created with storage backed by external non-HDF5 files (GH1000).
- Lists or tuples of unicode strings can now be stored as HDF5 attributes (GH1032).
- Inspecting the view returned by `.keys()` now shows the key names, for convenient interactive use (GH1049).

Exposing HDF5 functions

- `H5LTopen_file_image` as `h5py.h5f.open_file_image()` (GH1075).
- External dataset storage functions `H5Pset_external`, `H5Pget_external` and `H5Pget_external_count` as methods on `h5py.h5p.PropDCID` (GH1000).

Bugfixes

- Fix reading/writing of float128 data (GH1114).
- Converting data to float16 when creating a dataset (GH1115).

Support for old Python

Support for Python 3.3 has been dropped.

Support for Python 2.6 has been dropped.

6.1.14 What's new in h5py 2.8

This is the first release of the h5py 2.8 series. Note that the 2.8 series is the last series of h5py to support Python 2.6 and 3.3. Users should look to moving to Python 2.7 or (preferably) Python 3.4 or higher, as earlier releases are now outside of security support.

API changes

- Deprecation of `h5t.available_ftypes`. This is no longer used internally and will be removed in the future. There is no replacement public API. See GH926 for how to add addition floating point types to h5py.
- Do not sort fields in compound types (GH970 by James Tocknell). This is to account for changes in numpy 1.14.
- Minimum required version of Cython is now 0.23.

Features

- Allow registration of new file drivers (GH956 by Joe Jevnik).
- Add option to track object creation order to `Group.create_group` (GH968 by Chen Yufei)

Bug fixes

- Support slices with stop < start as empty slices (GH924 by Joe Jevnik)
- Avoid crashing the IPython auto-completer when missing key (GH885, GH958 by James Tocknell). The auto-completer currently only works on older versions of IPython, see GH1022 for what's needed to support newer versions of IPython/jupyter (PRs welcome!)
- Set libver default to 'earliest' (a.k.a LIBVER_EARLIEST) as previously documented (GH933, GH936 by James Tocknell)
- Fix conflict between fletcher32 and szip compression when using the float64 dtype (GH953, GH989, by Paul Müller).
- Update floating point type handling as flagged by numpy deprecation warning (GH985, by Eric Larson)
- Allow ExternalLinks to use non-ASCII hdf5 paths (GH333, GH952 by James Tocknell)
- Prefer custom HDF5 over pkg-config/fallback paths when building/installing (GH946, GH947 by Lars Viklund)
- Fix compatibility with Python 3 in document generation (GH921 by Ghislain Antony Vaillant)
- Fix spelling and grammar in documentation (GH931 by Michael V. DePalatis, GH950 by Christian Sachs, GH1015 by Mikhail)
- Add minor changes to documentation in order to improve clarity and warn about potential problems (GH528, GH783, GH829, GH849, GH911, GH959, by James Tocknell)
- Add license field to `setup.py` metadata (GH999 by Nils Werner).
- Use system encoding for errors, not utf-8 (GH1016, GH1025 by James Tocknell)
- Add `write_direct` to the documentation (GH1028 by Sajid Ali and Thomas A Caswell)

Wheels HDF5 Version

- Wheels uploaded to PyPI will now be built against the HDF5 1.10 series as opposed to the 1.8 series (h5py 2.8 is built against HDF5 1.10.2).

CI/Testing improvements and fixes

There were a number of improvements to testing and CI systems of h5py, including running the CI against multiple versions of HDF5, improving reliability and speed of the CIs, and simplifying the tox file. See GH857, GH894, GH922, GH954 and GH962 by Thomas A Caswell and James Tocknell for more details.

Other changes

- Emphasise reading from HDF5 files rather than writing to files in Quickguide (GH609, GH610 by Yu Feng). Note these changes were in the 2.5 branch, but never got merged into master. The h5py 2.8 release now actually includes these changes.
- Use lazy-loading of run_tests to avoid strong dependency on unittest2 (GH1013, GH1014 by Thomas VINCENT)
- Correctly handle with multiple float types of the same size (GH926 by James Tocknell)

Acknowledgements and Thanks

The h5py developers thank Nathan Goldbaum, Matthew Brett, and Christoph Gohlke for building the wheels that appear on PyPI.

6.1.15 What's new in h5py 2.7.1

2.7.1 is the first bug-fix release in the 2.7.x series.

Bug fixes

- GH903 Fixed critical issue with cyclic gc which resulted in segfaults
- GH904 Avoid unaligned access fixing h5py on sparc64
- GH883 Fixed compilation issues for some library locations
- GH868 Fix deadlock between phil and the import lock in py2
- GH841 Improve windows handling if filenames
- GH874 Allow close to be called on file multiple times
- GH867, GH872 Warn on loaded vs complied hdf5 version issues
- GH902 Fix overflow computing size of dataset on windows
- GH912 Do not mangle capitalization of filenames in error messages
- GH842 Fix longdouble on ppc64le
- GH862, GH916 Fix compounds structs with variable-size members

Fix h5py segfaulting on some Python 3 versions

Through an intersection of Python Issue 30484 and GH888, it was possible for the Python Garbage Collector to activate when closing `h5py` objects, which due to how dictionaries were iterated over in Python could cause a segfault. GH903 fixes the Garbage Collector activating whilst closing, whilst Python Issue 30484 had been fixed upstream (and backported to Python 3.3 onwards).

Avoid unaligned memory access in conversion functions

Some architectures (e.g. SPARC64) do not allow unaligned memory access, which can come up when copying packed structs. GH904 (by James Clarke) uses `memcpy` to avoid said unaligned memory access.

6.1.16 What's new in h5py 2.7

Python 3.2 is no longer supported

`h5py` 2.7 drops Python 3.2 support, and testing is not longer performed on Python 3.2. The latest versions of `pip`, `virtualenv`, `setuptools` and `numpy` do not support Python 3.2, and dropping 3.2 allows both `u` and `b` prefixes to be used for strings. A clean up of some of the legacy code was done in PR 675 by Andrew Collette.

Additionally, support for Python 2.6 is soon to be dropped for `pip` (See https://github.com/pypa/pip/issues/3955) and `setuptools` (See https://github.com/pypa/setuptools/issues/878), and `numpy` has dropped Python 2.6 also in the latest release. While `h5py` has not dropped Python 2.6 this release, users are strongly encouraged to move to Python 2.7 where possible.

Improved testing support

There has been a major increase in the number of configurations `h5py` is automatically tested in, with Windows CI support added via Appveyor (PR 795, PR 798, PR 799 and PR 801 by James Tocknell) and testing of minimum requirements to ensure we still satisfy them (PR 703 by James Tocknell). Additionally, `tox` was used to ensure that we don't run tests on Python versions which our dependencies have dropped or do not support (PR 662, PR 700 and PR 733). Thanks to to the Appveyor support, unicode tests were made more robust (PR 788, PR 800 and PR 804 by James Tocknell). Finally, other tests were improved or added where needed (PR 724 by Matthew Brett, PR 789, PR 794 and PR 802 by James Tocknell).

Improved python compatibility

The `ipython/jupyter` completion support now has Python 3 support (PR 715 by Joseph Kleinhenz). `h5py` now supports `pathlib` filenames (PR 716 by James Tocknell).

Documentation improvements

An update to the installation instructions and some whitespace cleanup was done in PR 808 by Thomas A Caswell, and mistake in the quickstart was fixed by Joydeep Bhattacharjee in PR 708.

setup.py improvements

Support for detecting the version of HDF5 via `pkgconfig` was added by Axel Huebl in PR 734, and support for specifying the path to MPI-supported HDF5 was added by Axel Huebl in PR 721. `h5py's` classifiers were updated to include supported python version and interpreters in PR 811 by James Tocknell.

Support for additional HDF5 features added

Low-level support for HDF5 Direct Chunk Write was added in PR 691 by Simon Gregor Ebner. Minimal support for HDF5 File Image Operations was added by Andrea Bedini in PR 680. Ideas and opinions for further support for both HDF5 Direct Chunk Write and HDF5 File Image Operations are welcome. High-level support for reading and writing null dataspaces was added in PR 664 by James Tocknell.

Improvements to type system

Reading and writing of compound datatypes has improved, with support for different orderings and alignments (PR 701 by Jonah Bernhard, PR 702 by Caleb Morse PR 738 by @smutch, PR 765 by Nathan Goldbaum and PR 793 by James Tocknell). Support for reading extended precision and non-standard floating point numbers has also been added (PR 749, PR 812 by Thomas A Caswell, PR 787 by James Tocknell and PR 781 by Martin Raspaud). Finally, compatibility improvements to `Cython` annotations of HDF5 types were added in PR 692 and PR 693 by Aleksandar Jelenak.

Other changes

- Fix deprecation of `-` for `numpy` boolean arrays (PR 683 by James Tocknell)
- Check for duplicates in fancy index validation (PR 739 by Sam Toyer)
- Avoid potential race condition (PR 754 by James Tocknell)
- Fix inconsistency when slicing with `numpy.array` of shape `(1,)` (PR 772 by Artsiom)
- Use `size_t` to store Python object id (PR 773 by Christoph Gohlke)
- Avoid errors when the Python GC runs during `nonlocal_close()` (PR 776 by Antoine Pitrou)
- Move from `six.PY3` to `six.PY2` (PR 686 by James Tocknell)

Acknowledgements

6.1.17 What's new in h5py 2.6

Support for HDF5 Virtual Dataset API

Initial support for the HDF5 Virtual Dataset API, which was introduced in HDF5 1.10, was added to the low-level API. Ideas and input for how this should work as part of the high-level interface are welcome.

This work was added in PR 663 by Aleksandar Jelenak.

Add MPI Collective I/O Support

Support for using MPI Collective I/O in both low-level and high-level code has been added. See the collective_io.py example for a simple demonstration of how to use MPI Collective I/O with the high level API.

This work was added in PR 648 by Jialin Liu.

Numerous build/testing/CI improvements

There were a number of improvements to the setup.py file, which should mean that *pip install h5py* should work in most places. Work was also done to clean up the current testing system, using tox is the recommended way of testing h5py across different Python versions. See PR 576 by Jakob Lombacher, PR 640 by Lawrence Mitchell, and PR 650, PR 651 and PR 658 by James Tocknell.

Cleanup of codebase based on pylint

There was a large cleanup of pylint-identified problems by Andrew Collette (PR 578, PR 579).

Fixes to low-level API

Fixes to the typing of functions were added in PR 597 by Ulrik Kofoed Pedersen, PR 589 by Peter Chang, and PR 625 by Spaghetti Sort. A fix for variable-length arrays was added in PR 621 by Sam Mason. Fixes to compound types were added in PR 639 by @nevion and PR 606 by Yu Feng. Finally, a fix to type conversion was added in PR 614 by Andrew Collette.

Documentation improvements

- Updates to FAQ by Dan Guest (PR 608) and Peter Hill (PR 607).
- Updates MPI-related documentation by Jens Timmerman (PR 604) and Matthias König (PR 572).
- Fixes to documentation building by Ghislain Antony Vaillant (PR 562, PR 561).
- Update PyTables link (PR 574 by Dominik Kriegner)
- Add File opening modes to docstring (PR 563 by Antony Lee)

Other changes

- Add *Dataset.ndim* (PR 649, PR 660 by @jakirkham, PR 661 by James Tocknell)
- Fix import errors in IPython completer (PR 605 by Niru Maheswaranathan)
- Turn off error printing in new threads (PR 583 by Andrew Collette)
- Use item value in *KeyError* instead of error message (PR 642 by Matthias Geier)

Acknowledgements

6.1.18 What's new in h5py 2.5

Experimental support for Single Writer Multiple Reader (SWMR)

This release introduces experimental support for the highly-anticipated "Single Writer Multiple Reader" (SWMR) feature in the upcoming HDF5 1.10 release. SWMR allows sharing of a single HDF5 file between multiple processes without the complexity of MPI or multiprocessing-based solutions.

This is an experimental feature that should NOT be used in production code. We are interested in getting feedback from the broader community with respect to performance and the API design.

For more details, check out the h5py user guide: https://docs.h5py.org/en/latest/swmr.html

SWMR support was contributed by Ulrik Pedersen (PR 551).

Other changes

- Use system Cython as a fallback if *cythonize()* fails (PR 541 by Ulrik Pedersen).
- Use pkg-config for building/linking against hdf5 (PR 505 by James Tocknell).
- Disable building Cython on Travis (PR 513 by Andrew Collette).
- Improvements to release tarball (PR 555, PR 560 by Ghislain Antony Vaillant).
- h5py now has one codebase for both Python 2 and 3; 2to3 removed from setup.py (PR 508 by James Tocknell).
- Add python 3.4 to tox (PR 507 by James Tocknell).
- Warn when importing from inside install dir (PR 558 by Andrew Collette).
- Tweak installation docs with reference to Anaconda and other Python package managers (PR 546 by Andrew Collette).
- Fix incompatible function pointer types (PR 526, PR 524 by Peter H. Li).
- Add explicit *vlen is not None* check to work around https://github.com/numpy/numpy/issues/2190 (*#538* by Will Parkin).
- Group and AttributeManager classes now inherit from the appropriate ABCs (PR 527 by James Tocknell).
- Don't strip metadata from special dtypes on read (PR 512 by Antony Lee).
- Add 'x' mode as an alias for 'w-' (PR 510 by Antony Lee).
- Support dynamical loading of LZF filter plugin (PR 506 by Peter Colberg).
- Fix accessing attributes with array type (PR 501 by Andrew Collette).
- Don't leak types in enum converter (PR 503 by Andrew Collette).
- Cython warning cleanups related to "const"

Acknowledgements

This release incorporates changes from, among others:

- Ulrik Pedersen
- James Tocknell
- Will Parkin
- Antony Lee
- Peter H. Li
- Peter Colberg
- Ghislain Antony Vaillant

6.1.19 What's new in h5py 2.4

Build system changes

The setup.py-based build system has been reworked to be more maintainable, and to fix certain long-standing bugs. As a consequence, the options to setup.py have changed; a new top-level "configure" command handles options like `--hdf5=/path/to/hdf5` and `--mpi`. Setup.py now works correctly under Python 3 when these options are used.

Cython (0.17+) is now required when building from source on all platforms; the .c files are no longer shipped in the UNIX release. The minimum NumPy version is now 1.6.1.

Files will now auto-close

Files are now automatically closed when all objects within them are unreachable. Previously, if File.close() was not explicitly called, files would remain open and "leaks" were possible if the File object was lost.

Thread safety improvements

Access to all APIs, high- and low-level, are now protected by a global lock. The entire API is now believed to be thread-safe. Feedback and real-world testing is welcome.

External link improvements

External links now work if the target file is already open. Previously this was not possible because of a mismatch in the file close strengths.

Thanks to

Many people, but especially:

- Matthieu Brucher
- Laurence Hole
- John Tyree
- Pierre de Buyl
- Matthew Brett

6.1.20 What's new in h5py 2.3

Support for arbitrary vlen data

Variable-length data is *no longer restricted to strings*. You can use this feature to produce "ragged" arrays, whose members are 1D arrays of variable length.

The implementation of special types was changed to use the NumPy dtype "metadata" field. This change should be transparent, as access to special types is handled through `h5py.special_dtype` and `h5py.check_dtype`.

Improved exception messages

H5py has historically suffered from low-detail exception messages generated automatically by HDF5. While the exception types in 2.3 remain identical to those in 2.2, the messages have been substantially improved to provide more information as to the source of the error.

Examples:

```
ValueError: Unable to set extend dataset (Dimension cannot exceed the existing maximal␣
↪size (new: 100 max: 1))

IOError: Unable to open file (Unable to open file: name = 'x3', errno = 2, error message␣
↪= 'no such file or directory', flags = 0, o_flags = 0)

KeyError: "Unable to open object (Object 'foo' doesn't exist)"
```

Improved setuptools support

`setup.py` now uses `setup_requires` to make installation via pip friendlier.

Multiple low-level additions

Improved support for opening datasets via the low-level interface, by adding `H5Dopen2` and many new property-list functions.

Improved support for MPI features

Added support for retrieving the MPI communicator and info objects from an open file. Added boilerplate code to allow compiling cleanly against newer versions of mpi4py.

Readonly files can now be opened in default mode

When opening a read-only file with no mode flags, now defaults to opening the file on RO mode rather than raising an exception.

Single-step build for HDF5 on Windows

Building h5py on windows has typically been hamstrung by the need to build a compatible version of HDF5 first. A new Paver-based system located in the "windows" distribution directory allows single-step compilation of HDF5 with settings that are known to work with h5py.

For more, see:

https://github.com/h5py/h5py/tree/master/windows

Thanks to

- Martin Teichmann
- Florian Rathgerber
- Pierre de Buyl
- Thomas Caswell
- Andy Salnikov
- Darren Dale
- Robert David Grant
- Toon Verstraelen
- Many others who contributed bug reports

6.1.21 What's new in h5py 2.2

Support for Parallel HDF5

On UNIX platforms, you can now take advantage of MPI and Parallel HDF5. Cython, `mpi4py` and an MPI-enabled build of HDF5 are required.. See *Parallel HDF5* in the documentation for details.

Support for Python 3.3

Python 3.3 is now officially supported.

Mini float support (issue #141)

Two-byte floats (NumPy `float16`) are supported.

HDF5 scale/offset filter

The Scale/Offset filter added in HDF5 1.8 is now available.

Field indexing is now allowed when writing to a dataset (issue #42)

H5py has long supported reading only certain fields from a dataset:

```
>>> dset = f.create_dataset('x', (100,), dtype=np.dtype([('a', 'f'), ('b', 'i')]))
>>> out = dset['a', 0:100:10]
>>> out.dtype
dtype('float32')
```

Now, field names are also allowed when writing to a dataset:

```
>>> dset['a', 20:50] = 1.0
```

Region references preserve shape (issue #295)

Previously, region references always resulted in a 1D selection, even when 2D slicing was used:

```
>>> dset = f.create_dataset('x', (10, 10))
>>> ref = dset.regionref[0:5,0:5]
>>> out = dset[ref]
>>> out.shape
(25,)
```

Shape is now preserved:

```
>>> out = dset[ref]
>>> out.shape
(5, 5)
```

Additionally, the shape of both the target dataspace and the selection shape can be determined via new methods on the `regionref` proxy (now available on both datasets and groups):

```
>>> f.regionref.shape(ref)
(10, 10)
>>> f.regionref.selection(ref)
(5, 5)
```

Committed types can be linked to datasets and attributes

HDF5 supports "shared" named types stored in the file:

```
>>> f['name'] = np.dtype("int64")
```

You can now use these types when creating a new dataset or attribute, and HDF5 will "link" the dataset type to the named type:

```
>>> dset = f.create_dataset('int dataset', (10,), dtype=f['name'])
>>> f.attrs.create('int scalar attribute', shape=(), dtype=f['name'])
```

`move` method on Group objects

It's no longer necessary to move objects in a file by manually re-linking them:

```
>>> f.create_group('a')
>>> f['b'] = f['a']
>>> del f['a']
```

The method `Group.move` allows this to be performed in one step:

```
>>> f.move('a', 'b')
```

Both the source and destination must be in the same file.

6.1.22 What's new in h5py 2.1

Dimension scales

H5py now supports the Dimension Scales feature of HDF5! Thanks to Darren Dale for implementing this. You can find more information on using scales in the *Dimension Scales* section of the docs.

Unicode strings allowed in attributes

Group, dataset and attribute names in h5py 2.X can all be given as unicode. Now, you can also store (scalar) unicode data in attribute values as well:

```
>>> myfile.attrs['x'] = u"I'm a Unicode string!"
```

Storing Unicode strings in datasets or as members of compound types is not yet implemented.

Dataset size property

Dataset objects now expose a `.size` property which provides the total number of elements in the dataspace.

`Dataset.value` property is now deprecated.

The property `Dataset.value`, which dates back to h5py 1.0, is deprecated and will be removed in a later release. This property dumps the entire dataset into a NumPy array. Code using `.value` should be updated to use NumPy indexing, using `mydataset[...]` or `mydataset[()]` as appropriate.

Bug fixes

- Object and region references were sometimes incorrectly wrapped wrapped in a `numpy.object_` instance (issue 202)
- H5py now ignores old versions of Cython (<0.13) when building (issue 221)
- Link access property lists weren't being properly tracked in the high level interface (issue 212)
- Race condition fixed in identifier tracking which led to Python crashes (issue 151)
- Highlevel objects will now complain if you try to bind them to the wrong HDF5 object types (issue 191)

- Unit tests can now be run after installation (issue 201)

6.1.23 What's new in h5py 2.0

HDF5 for Python (h5py) 2.0 represents the first major refactoring of the h5py codebase since the project's launch in 2008. Many of the most important changes are behind the scenes, and include changes to the way h5py interacts with the HDF5 library and Python. These changes have substantially improved h5py's stability, and make it possible to use more modern versions of HDF5 without compatibility concerns. It is now also possible to use h5py with Python 3.

Enhancements unlikely to affect compatibility

- HDF5 1.8.3 through 1.8.7 now work correctly and are officially supported.
- Python 3.2 is officially supported by h5py! Thanks especially to Darren Dale for getting this working.
- Fill values can now be specified when creating a dataset. The fill time is H5D_FILL_TIME_IFSET for contiguous datasets, and H5D_FILL_TIME_ALLOC for chunked datasets.
- On Python 3, dictionary-style methods like Group.keys() and Group.values() return view-like objects instead of lists.
- Object and region references now work correctly in compound types.
- Zero-length dimensions for extendable axes are now allowed.
- H5py no longer attempts to auto-import ipython on startup.
- File format bounds can now be given when opening a high-level File object (keyword "libver").

Changes which may break existing code

Supported HDF5/Python versions

- HDF5 1.6.X is no longer supported on any platform; following the release of 1.6.10 some time ago, this branch is no longer maintained by The HDF Group.
- Python 2.6 or later is now required to run h5py. This is a consequence of the numerous changes made to h5py for Python 3 compatibility.
- On Python 2.6, unittest2 is now required to run the test suite.

Group, Dataset and Datatype constructors have changed

In h5py 2.0, it is no longer possible to create new groups, datasets or named datatypes by passing names and settings to the constructors directly. Instead, you should use the standard Group methods create_group and create_dataset.

The File constructor remains unchanged and is still the correct mechanism for opening and creating files.

Code which manually creates Group, Dataset or Datatype objects will have to be modified to use create_group or create_dataset. File-resident datatypes can be created by assigning a NumPy dtype to a name (e.g. mygroup["name"] = numpy.dtype('S10')).

Unicode is now used for object names

Older versions of h5py used byte strings to represent names in the file. Starting with version 2.0, you may use either byte or unicode strings to create objects, but object names (obj.name, etc) will generally be returned as Unicode.

Code which may be affected:

- Anything which uses "isinstance" or explicit type checks on names, expecting "str" objects. Such checks should be removed, or changed to compare to "basestring" instead.
- In Python 2.X, other parts of your application may complain if they are handed Unicode data which can't be encoded down to ascii. This is a general problem in Python 2.

File objects must be manually closed

With h5py 1.3, when File objects (or low-level FileID) objects went out of scope, the corresponding HDF5 file was closed. This led to surprising behavior, especially when files were opened with the H5F_CLOSE_STRONG flag; "losing" the original File object meant that all open groups and datasets suddenly became invalid.

Beginning with h5py 2.0, files must be manually closed, by calling the "close" method or by using the file object as a context manager. If you forget to close a file, the HDF5 library will try to close it for you when the application exits.

Please note that opening the same file multiple times (i.e. without closing it first) continues to result in undefined behavior.

Changes to scalar slicing code

When a scalar dataset was accessed with the syntax `dataset[()]`, h5py incorrectly returned an ndarray. H5py now correctly returns an array scalar. Using `dataset[...]` on a scalar dataset still returns an ndarray.

Array scalars now always returned when indexing a dataset

When using datasets of compound type, retrieving a single element incorrectly returned a tuple of values, rather than an instance of `numpy.void_` with the proper fields populated. Among other things, this meant you couldn't do things like `dataset[index][field]`. H5py now always returns an array scalar, except in the case of object dtypes (references, vlen strings).

Reading object-like data strips special type information

In the past, reading multiple data points from dataset with vlen or reference type returned a Numpy array with a "special dtype" (such as those created by `h5py.special_dtype()`). In h5py 2.0, all such arrays now have a generic Numpy object dtype (`numpy.dtype('O')`). To get a copy of the dataset's dtype, always use the dataset's dtype property directly (`mydataset.dtype`).

The selections module has been removed

Only numpy-style slicing arguments remain supported in the high level interface. Existing code which uses the selections module should be refactored to use numpy slicing (and `numpy.s_` as appropriate), or the standard C-style HDF5 dataspace machinery.

The H5Error exception class has been removed (along with h5py.h5e)

All h5py exceptions are now native Python exceptions, no longer inheriting from H5Error. RuntimeError is raised if h5py can't figure out what exception is appropriate... every instance of this behavior is considered a bug. If you see h5py raising RuntimeError please report it so we can add the correct mapping!

The old errors module (h5py.h5e) has also been removed. There is no public error-management API.

File .mode property is now either 'r' or 'r+

Files can be opened using the same mode arguments as before, but now the property File.mode will always return 'r' (read-only) or 'r+' (read-write).

Long-deprecated dict methods have been removed

Certain ancient aliases for Group/AttributeManager methods (e.g. `listnames`) have been removed. Please use the standard Python dict interface (Python 2 or Python 3 as appropriate) to interact with these objects.

Known issues

- Thread support has been improved in h5py 2.0. However, we still recommend that for your own sanity you use locking to serialize access to files.
- There are reports of crashes related to storing object and region references. If this happens to you, please post on the mailing list or contact the h5py author directly.

6.2 Bug Reports & Contributions

Contributions and bug reports are welcome from anyone! Some of the best features in h5py, including thread support, dimension scales, and the scale-offset filter, came from user code contributions.

Since we use GitHub, the workflow will be familiar to many people. If you have questions about the process or about the details of implementing your feature, feel free to ask on Github itself, or on the h5py section of the HDF5 forum:

> https://forum.hdfgroup.org/c/hdf-tools/h5py

Posting on this forum requires registering for a free account with HDF group.

Anyone can post to this list. Your first message will be approved by a moderator, so don't worry if there's a brief delay.

This guide is divided into three sections. The first describes how to file a bug report.

The second describes the mechanics of how to submit a contribution to the h5py project; for example, how to create a pull request, which branch to base your work on, etc. We assume you're are familiar with Git, the version control system used by h5py. If not, here's a great place to start.

Finally, we describe the various subsystems inside h5py, and give technical guidance as to how to implement your changes.

6.2.1 How to File a Bug Report

Bug reports are always welcome! The issue tracker is at:

https://github.com/h5py/h5py/issues

If you're unsure whether you've found a bug

Always feel free to ask on the mailing list (h5py at Google Groups). Discussions there are seen by lots of people and are archived by Google. Even if the issue you're having turns out not to be a bug in the end, other people can benefit from a record of the conversation.

By the way, nobody will get mad if you file a bug and it turns out to be something else. That's just how software development goes.

What to include

When filing a bug, there are two things you should include. The first is the output of `h5py.version.info`:

```
>>> import h5py
>>> print(h5py.version.info)
```

The second is a detailed explanation of what went wrong. Unless the bug is really trivial, **include code if you can**, either via GitHub's inline markup:

```
```
    import h5py
    h5py.explode()   # Destroyed my computer!
```
```

or by uploading a code sample to Github Gist.

6.2.2 How to Get Your Code into h5py

This section describes how to contribute changes to the h5py code base. Before you start, be sure to read the h5py license and contributor agreement in "license.txt". You can find this in the source distribution, or view it online at the main h5py repository at GitHub.

The basic workflow is to clone h5py with git, make your changes in a topic branch, and then create a pull request at GitHub asking to merge the changes into the main h5py project.

Here are some tips to getting your pull requests accepted:

1. Let people know you're working on something. This could mean posting a comment in an open issue, or sending an email to the mailing list. There's nothing wrong with just opening a pull request, but it might save you time if you ask for advice first.
2. Keep your changes focused. If you're fixing multiple issues, file multiple pull requests. Try to keep the amount of reformatting clutter small so the maintainers can easily see what you've changed in a diff.
3. Unit tests are mandatory for new features. This doesn't mean hundreds (or even dozens) of tests! Just enough to make sure the feature works as advertised. The maintainers will let you know if more are needed.

Clone the h5py repository

The best way to do this is by signing in to GitHub and cloning the h5py project directly. You'll end up with a new repository under your account; for example, if your username is `yourname`, the repository would be at http://github.com/yourname/h5py.

Then, clone your new copy of h5py to your local machine:

```
$ git clone http://github.com/yourname/h5py
```

Create a topic branch for your feature

Check out a new branch for the bugfix or feature you're writing:

```
$ git checkout -b newfeature master
```

The exact name of the branch can be anything you want. For bug fixes, one approach is to put the issue number in the branch name.

We develop all changes against the *master* branch. If we're making a bugfix release, a bot will backport merged pull requests.

Implement the feature!

You can implement the feature as a number of small changes, or as one big commit; there's no project policy. Double-check to make sure you've included all your files; run `git status` and check the output.

Run the tests

The easiest way to run the tests is with tox:

```
pip install tox  # Get tox

tox -e py37-test-deps  # Run tests in one environment
tox                    # Run tests in all possible environments
tox -a                 # List defined environments
```

Write a release note

Changes which could affect people building and using h5py after the next release should have a news entry. You don't need to do this if your changes don't affect usage, e.g. adding tests or correcting comments.

In the `news/` folder, make a copy of `TEMPLATE.rst` named after your branch. Edit the new file, adding a sentence or two about what you've added or fixed. Commit this to git too.

News entries are merged into the *what's new documents* for each release. They should allow someone to quickly understand what a new feature is, or whether a bug they care about has been fixed. E.g.:

```
Bug fixes
---------

* Fix reading data for region references pointing to an empty selection.
```

The *Building h5py* section is for changes which affect how people build h5py from source. It's not about how we make prebuilt wheels; changes to that which make a visible difference can go in *New features* or *Bug fixes*.

Push your changes back and open a pull request

Push your topic branch back up to your GitHub clone:

```
$ git push origin newfeature
```

Then, create a pull request based on your topic branch.

Work with the maintainers

Your pull request might be accepted right away. More commonly, the maintainers will post comments asking you to fix minor things, like add a few tests, clean up the style to be PEP-8 compliant, etc.

The pull request page also shows the results of building and testing the modified code on Travis and Appveyor CI and Azure Pipelines. Check back after about 30 minutes to see if the build succeeded, and if not, try to modify your changes to make it work.

When making changes after creating your pull request, just add commits to your topic branch and push them to your GitHub repository. Don't try to rebase or open a new pull request! We don't mind having a few extra commits in the history, and it's helpful to keep all the history together in one place.

6.2.3 How to Modify h5py

This section is a little more involved, and provides tips on how to modify h5py. The h5py package is built in layers. Starting from the bottom, they are:

1. The HDF5 C API (provided by libhdf5)
2. Auto-generated Cython wrappers for the C API (`api_gen.py`)
3. Low-level interface, written in Cython, using the wrappers from (2)
4. High-level interface, written in Python, with things like `h5py.File`.
5. Unit test code

Rather than talk about the layers in an abstract way, the parts below are guides to adding specific functionality to various parts of h5py. Most sections span at least two or three of these layers.

Adding a function from the HDF5 C API

This is one of the most common contributed changes. The example below shows how one would add the function `H5Dget_storage_size`, which determines the space on disk used by an HDF5 dataset. This function is already partially wrapped in h5py, so you can see how it works.

It's recommended that you follow along, if not by actually adding the feature then by at least opening the various files as we work through the example.

First, get ahold of the function signature; the easiest place for this is at the online HDF5 Reference Manual. Then, add the function's C signature to the file `api_functions.txt`:

```
hsize_t   H5Dget_storage_size(hid_t dset_id)
```

This particular signature uses types (`hsize_t`, `hid_t`) which are already defined elsewhere. But if the function you're adding needs a struct or enum definition, you can add it using Cython code to the file `api_types_hdf5.pxd`.

The next step is to add a Cython function or method which calls the function you added. The h5py modules follow the naming convention of the C API; functions starting with `H5D` are wrapped in `h5d.pyx`.

Opening `h5d.pyx`, we notice that since this function takes a dataset identifier as the first argument, it belongs as a method on the DatasetID object. We write a wrapper method:

```
def get_storage_size(self):
    """ () => LONG storage_size

        Determine the amount of file space required for a dataset.  Note
        this only counts the space which has actually been allocated; it
        may even be zero.
    """
    return H5Dget_storage_size(self.id)
```

The first line of the docstring gives the method signature. This is necessary because Cython will use a "generic" signature like `method(*args, **kwds)` when the file is compiled. The h5py documentation system will extract the first line and use it as the signature.

Next, we decide whether we want to add access to this function to the high-level interface. That means users of the top-level `h5py.Dataset` object will be able to see how much space on disk their files use. The high-level interface is implemented in the subpackage `h5py._hl`, and the Dataset object is in module `dataset.py`. Opening it up, we add a property on the `Dataset` object:

```
@property
def storagesize(self):
    """ Size (in bytes) of this dataset on disk. """
    return self.id.get_storage_size()
```

You'll see that the low-level `DatasetID` object is available on the high-level `Dataset` object as `obj.id`. This is true of all the high-level objects, like `File` and `Group` as well.

Finally (and don't skip this step), we write **unit tests** for this feature. Since the feature is ultimately exposed at the high-level interface, it's OK to write tests for the `Dataset.storagesize` property only. Unit tests for the high-level interface are located in the "tests" subfolder, right near `dataset.py`.

It looks like the right file is `test_dataset.py`. Unit tests are implemented as methods on custom `unittest.UnitTest` subclasses; each new feature should be tested by its own new class. In the `test_dataset` module, we see there's already a subclass called `BaseDataset`, which implements some simple set-up and cleanup methods and provides a `h5py.File` object as `obj.f`. We'll base our test class on that:

```
class TestStorageSize(BaseDataset):

    """
        Feature: Dataset.storagesize indicates how much space is used.
    """

    def test_empty(self):
        """ Empty datasets take no space on disk """
        dset = self.f.create_dataset("x", (100,100))
        self.assertEqual(dset.storagesize, 0)

    def test_data(self):
```

(continues on next page)

(continued from previous page)

```
        """ Storage size is correct for non-empty datasets """
        dset = self.f.create_dataset("x", (100,), dtype='uint8')
        dset[...] = 42
        self.assertEqual(dset.storagesize, 100)
```

This set of tests would be adequate to get a pull request approved. We don't test every combination under the sun (different ranks, datasets with more than 2**32 elements, datasets with the string "kumquat" in the name...), but the basic, commonly encountered set of conditions.

To build and test our changes, we have to do a few things. First of all, run the file `api_gen.py` to re-generate the Cython wrappers from `api_functions.txt`:

```
$ python api_gen.py
```

Then build the project, which recompiles `h5d.pyx`:

```
$ python setup.py build
```

Finally, run the test suite, which includes the two methods we just wrote:

```
$ python setup.py test
```

If the tests pass, the feature is ready for a pull request.

Adding a function only available in certain versions of HDF5

At the moment, h5py must be compatible with HDF5 back to version 1.10.4. But it's possible to conditionally include functions which only appear in newer versions of HDF5. It's also possible to mark functions which require Parallel HDF5. For example, the function `H5Fset_mpi_atomicity` was introduced in HDF5 1.8.9 and requires Parallel HDF5. Specifiers before the signature in `api_functions.txt` communicate this:

```
MPI 1.8.9 herr_t H5Fset_mpi_atomicity(hid_t file_id, hbool_t flag)
```

You can specify either, both or none of "MPI" or a version number in "X.Y.Z" format.

In the Cython code, these show up as "preprocessor" defines `MPI` and `HDF5_VERSION`. So the low-level implementation (as a method on `h5py.h5f.FileID`) looks like this:

```
IF MPI and HDF5_VERSION >= (1, 8, 9):

    def set_mpi_atomicity(self, bint atomicity):
        """ (BOOL atomicity)

        For MPI-IO driver, set to atomic (True), which guarantees sequential
        I/O semantics, or non-atomic (False), which improves  performance.

        Default is False.

        Feature requires: 1.8.9 and Parallel HDF5
        """
        H5Fset_mpi_atomicity(self.id, <hbool_t>atomicity)
```

High-level code can check the version of the HDF5 library, or check to see if the method is present on `FileID` objects.

Testing MPI-only features/code

Typically to run code under MPI, `mpirun` must be used to start the MPI processes. Similarly, tests using MPI features (such as collective IO), must also be run under `mpirun`. h5py uses pytest markers (specifically `pytest.mark.mpi` and other markers from pytest-mpi) to specify which tests require usage of `mpirun`, and will handle skipping the tests as needed. A simple example of how to do this is:

```
@pytest.mark.mpi
def test_mpi_feature():
   import mpi4py
   # test the MPI feature
```

To run these tests, you'll need to:

1. Have `tox` installed (e.g. via `pip install tox`)
2. Have HDF5 built with MPI as per *Building against Parallel HDF5*

Then running:

```
$ CC='mpicc' HDF5_MPI=ON tox -e py37-test-deps-mpi4py
```

should run the tests. You may need to pass `HDF5_DIR` depending on the location of the HDF5 with MPI support. You can choose which python version to build against by changing py37 (e.g. py36 runs python 3.6, this is a tox feature), and test with the minimum version requirements by using `mindeps` rather than `deps`.

If you get an error similar to:

```
There are not enough slots available in the system to satisfy the 4 slots
that were requested by the application:
  python

Either request fewer slots for your application, or make more slots available
for use.
```

then you need to reduce the number of MPI processes you are asking MPI to use. If you have already reduced the number of processes requested (or are running the default number which is 2), you will need to look up the documentation for your MPI implementation for handling this error. On OpenMPI (which is usually the default MPI implementation on most systems), running:

```
$ export OMPI_MCA_rmaps_base_oversubscribe=1
```

will instruct OpenMPI to allow more MPI processes than available cores on your system.

If you need to pass additional environment variables to your MPI implementation, add these variables to the `passenv` setting in the `tox.ini`, and send us a PR with that change noting the MPI implementation.

6.3 Release Guide

h5py uses rever for release management. To install rever, use either pip or conda:

```
# pip
$ pip install re-ver

# conda
$ conda install -c conda-forge rever
```

6.3.1 Performing releases

Once rever is installed, always run the `check` command to make sure that everything you need to perform the release is correctly installed and that you have the correct permissions. All rever commands should be run in the root level of the repository.

Step 1 (repeat until successful)

```
$ rever check
```

Resolve any issues that may have come up, and keep running `rever check` until it passes. After it is successful, simply pass the version number you want to release (e.g. `X.Y.Z`) into the rever command.

Step 2

```
$ rever X.Y.Z
```

You probably want to make sure (with `git tag`) that the new version number is available. If any release activities fail while running this command, you may safely re-run this command. You can also safely undo previously run activities. Please see the rever docs for more details.

6.4 FAQ

6.4.1 What datatypes are supported?

Below is a complete list of types for which h5py supports reading, writing and creating datasets. Each type is mapped to a native NumPy type.

Fully supported types:

Type	Precisions	Notes
Bitfield	1, 2, 4 or 8 byte, BE/LE	Read as unsigned integers
Integer	1, 2, 4 or 8 byte, BE/LE, signed/unsigned	
Float	2, 4, 8, 12, 16 byte, BE/LE	
Complex	8 or 16 byte, BE/LE	Stored as HDF5 struct
Compound	Arbitrary names and offsets	
Strings (fixed-length)	Any length	
Strings (variable-length)	Any length, ASCII or Unicode	
Opaque (kind 'V')	Any length	
Boolean	NumPy 1-byte bool	Stored as HDF5 enum
Array	Any supported type	
Enumeration	Any NumPy integer type	Read/write as integers
References	Region and object	
Variable length array	Any supported type	See *Special Types*

Other numpy dtypes, such as datetime64 and timedelta64, can optionally be stored in HDF5 opaque data using `opaque_dtype()`. h5py will read this data back with the same dtype, but other software probably will not understand it.

Unsupported types:

Type	Status
HDF5 "time" type	
NumPy "U" strings	No HDF5 equivalent
NumPy generic "O"	Not planned

6.4.2 What compression/processing filters are supported?

Filter	Function	Availability
DEFLATE/GZIP	Standard HDF5 compression	All platforms
SHUFFLE	Increase compression ratio	All platforms
FLETCHER32	Error detection	All platforms
Scale-offset	Integer/float scaling and truncation	All platforms
SZIP	Fast, patented compression for int/float	• UNIX: if supplied with HDF5. • Windows: read-only
LZF	Very fast compression, all types	Ships with h5py, C source available

6.4.3 What file drivers are available?

A number of different HDF5 "drivers", which provide different modes of access to the filesystem, are accessible in h5py via the high-level interface. The currently supported drivers are:

Driver	Purpose	Notes
sec2	Standard optimized driver	Default on UNIX/Windows
stdio	Buffered I/O using stdio.h	
core	In-memory file (optionally backed to disk)	
family	Multi-file driver	
mpio	Parallel HDF5 file access	

6.4.4 What's the difference between h5py and PyTables?

The two projects have different design goals. PyTables presents a database-like approach to data storage, providing features like indexing and fast "in-kernel" queries on dataset contents. It also has a custom system to represent data types.

In contrast, h5py is an attempt to map the HDF5 feature set to NumPy as closely as possible. For example, the high-level type system uses NumPy dtype objects exclusively, and method and attribute naming follows Python and NumPy conventions for dictionary and array access (i.e. ".dtype" and ".shape" attributes for datasets, `group[name]` indexing syntax for groups, etc).

Underneath the "high-level" interface to h5py (i.e. NumPy-array-like objects; what you'll typically be using) is a large Cython layer which calls into C. This "low-level" interface provides access to nearly all of the HDF5 C API. This layer is object-oriented with respect to HDF5 identifiers, supports reference counting, automatic translation between NumPy and HDF5 type objects, translation between the HDF5 error stack and Python exceptions, and more.

This greatly simplifies the design of the complicated high-level interface, by relying on the "Pythonicity" of the C API wrapping.

There's also a PyTables perspective on this question at the PyTables FAQ.

6.4.5 Does h5py support Parallel HDF5?

Starting with version 2.2, h5py supports Parallel HDF5 on UNIX platforms. `mpi4py` is required, as well as an MPIO-enabled build of HDF5. Check out *Parallel HDF5* for details.

6.4.6 Variable-length (VLEN) data

Starting with version 2.3, all supported types can be stored in variable-length arrays (previously only variable-length byte and unicode strings were supported) See *Special Types* for use details. Please note that since strings in HDF5 are encoded as ASCII or UTF-8, NUL bytes are not allowed in strings.

6.4.7 Enumerated types

HDF5 enumerated types are supported. As NumPy has no native enum type, they are treated on the Python side as integers with a small amount of metadata attached to the dtype.

6.4.8 NumPy object types

Storage of generic objects (NumPy dtype "O") is not implemented and not planned to be implemented, as the design goal for h5py is to expose the HDF5 feature set, not add to it. However, objects picked to the "plain-text" protocol (protocol 0) can be stored in HDF5 as strings.

6.4.9 Appending data to a dataset

The short response is that h5py is NumPy-like, not database-like. Unlike the HDF5 packet-table interface (and PyTables), there is no concept of appending rows. Rather, you can expand the shape of the dataset to fit your needs. For example, if I have a series of time traces 1024 points long, I can create an extendable dataset to store them:

```
>>> dset = myfile.create_dataset("MyDataset", (10, 1024), maxshape=(None, 1024))
>>> dset.shape
(10,1024)
```

The keyword argument "maxshape" tells HDF5 that the first dimension of the dataset can be expanded to any size, while the second dimension is limited to a maximum size of 1024. We create the dataset with room for an initial ensemble of 10 time traces. If we later want to store 10 more time traces, the dataset can be expanded along the first axis:

```
>>> dset.resize(20, axis=0)   # or dset.resize((20,1024))
>>> dset.shape
(20, 1024)
```

Each axis can be resized up to the maximum values in "maxshape". Things to note:

- Unlike NumPy arrays, when you resize a dataset the indices of existing data do not change; each axis grows or shrinks independently
- The dataset rank (number of dimensions) is fixed when it is created

6.4.10 Unicode

As of h5py 2.0.0, Unicode is supported for file names as well as for objects in the file. When object names are read, they are returned as Unicode by default.

However, HDF5 has no predefined datatype to represent fixed-width UTF-16 or UTF-32 (NumPy format) strings. Therefore, the NumPy 'U' datatype is not supported.

6.4.11 Exceptions

h5py tries to map the error codes from hdf5 to the corresponding `Exception` class on the Python side. However the HDF5 group does not consider the error codes to be public API so we can not guarantee type stability of the exceptions raised.

6.4.12 Development

Building from Git

We moved to GitHub in December of 2012 (http://github.com/h5py/h5py).

We use the following conventions for branches and tags:

- master: integration branch for the next minor (or major) version
- 2.0, 2.1, 2.2, etc: bugfix branches for released versions
- tags 2.0.0, 2.0.1, etc: Released bugfix versions

To build from a Git checkout:

Clone the project:

```
$ git clone https://github.com/h5py/h5py.git
$ cd h5py
```

(Optional) Choose which branch to build from (e.g. a stable branch):

```
$ git checkout 2.1
```

Build the project. If given, /path/to/hdf5 should point to a directory containing a compiled, shared-library build of HDF5 (containing things like "include" and "lib"):

```
$ python setup.py build [--hdf5=/path/to/hdf5]
```

(Optional) Run the unit tests:

```
$ python setup.py test
```

Report any failing tests to the mailing list (h5py at googlegroups), or by filing a bug report at GitHub.

6.5 Licenses and legal info

6.5.1 Copyright Notice and Statement for the h5py Project

```
Copyright (c) 2008 Andrew Collette and contributors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:
```

(continues on next page)

(continued from previous page)

```
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the
   distribution.

3. Neither the name of the copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

6.5.2 HDF5 Copyright Statement

```
HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 2006-2007 by The HDF Group (THG).

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998-2006 by the Board of Trustees of the University of Illinois.

All rights reserved.

Contributors: National Center for Supercomputing Applications (NCSA)
at the University of Illinois, Fortner Software, Unidata Program
Center (netCDF), The Independent JPEG Group (JPEG), Jean-loup Gailly
and Mark Adler (gzip), and Digital Equipment Corporation (DEC).

Redistribution and use in source and binary forms, with or without
modification, are permitted for any purpose (including commercial
purposes) provided that the following conditions are met:

   1. Redistributions of source code must retain the above copyright
notice, this list of conditions, and the following disclaimer.
   2. Redistributions in binary form must reproduce the above
copyright notice, this list of conditions, and the following
disclaimer in the documentation and/or materials provided with the
distribution.
```

(continues on next page)

(continued from previous page)

```
   3. In addition, redistributions of modified forms of the source or
binary code must carry prominent notices stating that the original
code was changed and the date of the change.
   4. All publications or advertising materials mentioning features or
use of this software are asked, but not required, to acknowledge that
it was developed by The HDF Group and by the National Center for
Supercomputing Applications at the University of Illinois at
Urbana-Champaign and credit the contributors.
   5. Neither the name of The HDF Group, the name of the University,
nor the name of any Contributor may be used to endorse or promote
products derived from this software without specific prior written
permission from THG, the University, or the Contributor, respectively.

DISCLAIMER: THIS SOFTWARE IS PROVIDED BY THE HDF GROUP (THG) AND THE
CONTRIBUTORS "AS IS" WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR
IMPLIED. In no event shall THG or the Contributors be liable for any
damages suffered by the users arising out of the use of this software,
even if advised of the possibility of such damage.

Portions of HDF5 were developed with support from the University of
California, Lawrence Livermore National Laboratory (UC LLNL). The
following statement applies to those portions of the product and must
be retained in any redistribution of source code, binaries,
documentation, and/or accompanying materials:

This work was partially produced at the University of California,
Lawrence Livermore National Laboratory (UC LLNL) under contract
no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy
(DOE) and The Regents of the University of California (University) for
the operation of UC LLNL.

DISCLAIMER: This work was prepared as an account of work sponsored by
an agency of the United States Government. Neither the United States
Government nor the University of California nor any of their
employees, makes any warranty, express or implied, or assumes any
liability or responsibility for the accuracy, completeness, or
usefulness of any information, apparatus, product, or process
disclosed, or represents that its use would not infringe privately-
owned rights. Reference herein to any specific commercial products,
process, or service by trade name, trademark, manufacturer, or
otherwise, does not necessarily constitute or imply its endorsement,
recommendation, or favoring by the United States Government or the
University of California. The views and opinions of authors expressed
herein do not necessarily state or reflect those of the United States
Government or the University of California, and shall not be used for
advertising or product endorsement purposes.
```

6.5.3 PyTables Copyright Statement

```
Copyright Notice and Statement for PyTables Software Library and Utilities:

Copyright (c) 2002, 2003, 2004  Francesc Altet
Copyright (c) 2005, 2006, 2007  Carabos Coop. V.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

a. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

b. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the
   distribution.

c. Neither the name of the Carabos Coop. V. nor the names of its
   contributors may be used to endorse or promote products derived
   from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

6.5.4 stdint.h (Windows version) License

```
Copyright (c) 2006-2008 Alexander Chemeris

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice,
     this list of conditions and the following disclaimer.

  2. Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

  3. The name of the author may be used to endorse or promote products
```

(continues on next page)

(continued from previous page)

```
     derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

6.5.5 Python license

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python Python 2.7.5 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python Python 2.7.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright 2001-2013 Python Software Foundation; All Rights Reserved" are retained in Python Python 2.7.5 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python Python 2.7.5 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python Python 2.7.5.
4. PSF is making Python Python 2.7.5 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON Python 2.7.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON Python 2.7.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON Python 2.7.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python Python 2.7.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

INDEX

Symbols

__bool__() (*h5py.Dataset method*), 38
__bool__() (*h5py.File method*), 22
__bool__() (*h5py.Group method*), 26
__contains__() (*h5py.AttributeManager method*), 42
__contains__() (*h5py.Group method*), 26
__delitem__() (*h5py.AttributeManager method*), 42
__getitem__() (*h5py.AttributeManager method*), 42
__getitem__() (*h5py.Dataset method*), 38
__getitem__() (*h5py.Group method*), 26
__iter__() (*h5py.AttributeManager method*), 42
__iter__() (*h5py.Group method*), 26
__setitem__() (*h5py.AttributeManager method*), 42
__setitem__() (*h5py.Dataset method*), 38
__setitem__() (*h5py.Group method*), 26

A

asstr() (*h5py.Dataset method*), 39
astype() (*h5py.Dataset method*), 38
AttributeManager (*class in h5py*), 42
attrs (*h5py.Dataset attribute*), 41
attrs (*h5py.Group attribute*), 30

B

build_virtual_dataset() (*h5py.Group method*), 30

C

check_dtype() (*in module h5py*), 51
check_enum_dtype() (*in module h5py*), 50
check_opaque_dtype() (*in module h5py*), 50
check_string_dtype() (*in module h5py*), 48
check_vlen_dtype() (*in module h5py*), 49
chunks (*h5py.Dataset attribute*), 40
close() (*h5py.File method*), 23
compression (*h5py.Dataset attribute*), 40
compression_opts (*h5py.Dataset attribute*), 40
copy() (*h5py.Group method*), 27
create() (*h5py.AttributeManager method*), 42
create_dataset() (*h5py.Group method*), 28
create_dataset_like() (*h5py.Group method*), 29
create_group() (*h5py.Group method*), 28
create_virtual_dataset() (*h5py.Group method*), 30

D

Dataset (*class in h5py*), 38
dims (*h5py.Dataset attribute*), 41
driver (*h5py.File attribute*), 23
dtype (*h5py.Dataset attribute*), 40

E

encoding (*h5py.string_info attribute*), 48
enum_dtype() (*in module h5py*), 49
external (*h5py.Dataset attribute*), 40
ExternalLink (*class in h5py*), 31

F

fields() (*h5py.Dataset method*), 39
File (*class in h5py*), 21
file (*h5py.Dataset attribute*), 41
file (*h5py.Group attribute*), 31
filename (*h5py.ExternalLink attribute*), 31
filename (*h5py.File attribute*), 23
fillvalue (*h5py.Dataset attribute*), 40
fletcher32 (*h5py.Dataset attribute*), 40
flush() (*h5py.File method*), 23

G

get() (*h5py.AttributeManager method*), 42
get() (*h5py.Group method*), 27
get_id() (*h5py.AttributeManager method*), 42
Group (*class in h5py*), 26

H

HardLink (*class in h5py*), 31

I

id (*h5py.Dataset attribute*), 41
id (*h5py.File attribute*), 23
id (*h5py.Group attribute*), 30
is_scale (*h5py.Dataset attribute*), 41
is_virtual (*h5py.Dataset attribute*), 40
items() (*h5py.AttributeManager method*), 42

items() (*h5py.Group method*), 26
iter_chunks() (*h5py.Dataset method*), 39

K

keys() (*h5py.AttributeManager method*), 42
keys() (*h5py.Group method*), 26

L

len() (*h5py.Dataset method*), 39
length (*h5py.string_info attribute*), 48
libver (*h5py.File attribute*), 23

M

make_scale() (*h5py.Dataset method*), 39
maxshape (*h5py.Dataset attribute*), 40
meta_block_size (*h5py.File attribute*), 23
mode (*h5py.File attribute*), 23
modify() (*h5py.AttributeManager method*), 43
move() (*h5py.Group method*), 27

N

name (*h5py.Dataset attribute*), 41
name (*h5py.Group attribute*), 30
nbytes (*h5py.Dataset attribute*), 40
ndim (*h5py.Dataset attribute*), 40

O

opaque_dtype() (*in module h5py*), 50

P

parent (*h5py.Dataset attribute*), 41
parent (*h5py.Group attribute*), 31
path (*h5py.ExternalLink attribute*), 31
path (*h5py.SoftLink attribute*), 31
Python Enhancement Proposals
    PEP 538, 19

R

read_direct() (*h5py.Dataset method*), 38
ref (*h5py.Dataset attribute*), 41
ref (*h5py.Group attribute*), 30
regionref (*h5py.Dataset attribute*), 41
regionref (*h5py.Group attribute*), 30
require_dataset() (*h5py.Group method*), 29
require_group() (*h5py.Group method*), 28
resize() (*h5py.Dataset method*), 39

S

scaleoffset (*h5py.Dataset attribute*), 40
shape (*h5py.Dataset attribute*), 40
shuffle (*h5py.Dataset attribute*), 40
size (*h5py.Dataset attribute*), 40
SoftLink (*class in h5py*), 31
special_dtype() (*in module h5py*), 50
string_dtype() (*in module h5py*), 48
string_info (*class in h5py*), 48
swmr_mode (*h5py.File attribute*), 23

U

userblock_size (*h5py.File attribute*), 23

V

values() (*h5py.AttributeManager method*), 42
values() (*h5py.Group method*), 26
virtual_sources() (*h5py.Dataset method*), 40
VirtualLayout (*class in h5py*), 65
VirtualSource (*class in h5py*), 66
visit() (*h5py.Group method*), 27
visititems() (*h5py.Group method*), 27
vlen_dtype() (*in module h5py*), 49

W

write_direct() (*h5py.Dataset method*), 38