h5py Documentation

Release 2.6.0

Andrew Collette and contributors

January 13, 2017

Contents

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 1

Where to start

- *Quick-start guide*
- *Installation*

CHAPTER 2

Other resources

- Python and HDF5 O'Reilly book
- Ask questions on the mailing list at Google Groups
- GitHub project

CHAPTER 3

Introductory info

3.1 Quick Start Guide

3.1.1 Install

With Anaconda or Miniconda:

```
conda install h5py
```

With Enthought Canopy, use the GUI package manager or:

```
enpkg h5py
```

With pip or setup.py, 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**

The very first thing you'll need to do is create a new file:

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

The *File object* is your starting point. It has a couple of methods which look interesting. One of them is `create_dataset`:

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

The object we created 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*.

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
u'/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
u'/'
```

Creating a subgroup is accomplished via the aptly-named `create_group`:

```
>>> 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
u'/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
u'/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
```

Containership 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

3.2.1 Pre-configured installation (recommended)

It's strongly recommended that you use a Python distribution or package manager to install h5py along with its compiled dependencies. Here are some which are popular in the Python community:

- Anaconda or Miniconda (Mac, Windows, Linux)
- Enthought Canopy (Mac, Windows, Linux)
- PythonXY (Windows)

```
conda install h5py  # Anaconda/Miniconda
enpkg h5py          # Canopy
```

Or, use your package manager:

- apt-get (Linux/Debian, including Ubuntu)

- yum (Linux/Red Hat, including Fedora and CentOS)
- Homebrew (OS X)
- pacman (Arch linux)

3.2.2 Source installation on Linux and OS X

You need, via apt-get, yum or Homebrew:

- Python 2.6, 2.7, 3.3, 3.4, or 3.5 with development headers (`python-dev` or similar)
- HDF5 1.8.4 or newer, shared library version with development headers (`libhdf5-dev` or similar)
- NumPy 1.6.1 or later

```
$ pip install h5py
```

or, from a tarball or git *checkout*

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

or

```
$ python setup.py install
```

If you are working on a development version and the underlying cython files change it may be necessary to force a full rebuild. The easiest way to achieve this is

```
$ git clean -xfd
```

from the top of your clone and then rebuilding.

3.2.3 Source installation on Windows

Installing from source on Windows is effectively impossible because of the C library dependencies involved.

If you don't want to use Anaconda, Canopy, or PythonXY, download a third-party wheel from Chris Gohlke's excellent collection.

3.2.4 Custom installation

You can specify build options for h5py with the `configure` option to setup.py. Options may be given together or separately:

```
$ python setup.py configure --hdf5=/path/to/hdf5
$ python setup.py configure --hdf5-version=X.Y.Z
$ python setup.py configure --mpi
```

Note the `--hdf5-version` option is generally not needed, as h5py auto-detects the installed version of HDF5 (even for custom locations).

Once set, build options apply to all future builds in the source directory. You can reset to the defaults with the `--reset` option:

```
$ python setup.py configure --reset
```

You can also configure h5py using environment variables. This is handy when installing via `pip`, as you don't have direct access to setup.py:

```
$ HDF5_DIR=/path/to/hdf5 pip install h5py
$ HDF5_VERSION=X.Y.Z pip install h5py
$ CC="mpicc" HDF5_MPI="ON" HDF5_DIR=/path/to/parallel-hdf5 pip install h5py
```

Here's a list of all the configure options currently supported:

Option	Via setup.py	Via environment variable
Custom path to HDF5	`--hdf5=/path/to/hdf5`	`HDF5_DIR=/path/to/hdf5`
Force HDF5 version	`--hdf5-version=X.Y.Z`	`HDF5_VERSION=X.Y.Z`
Enable MPI mode	`--mpi`	`HDF5_MPI=ON`

3.2.5 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
$ python setup.py install
```

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 either be done with command-line options from the h5py tarball or by:

```
$ export HDF5_MPI="ON"
```

You will need a shared-library build of Parallel HDF5 (i.e. built with ./configure –enable-shared –enable-parallel).

To build in MPI mode, use the `--mpi` option to `setup.py configure` or export `HDF5_MPI="ON"` beforehand:

```
$ export CC=mpicc
$ export HDF5_MPI="ON"
$ python setup.py configure
$ python setup.py build
```

See also *Parallel HDF5*.

CHAPTER 4

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
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 (default)

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' Memory-map the entire file; all operations are performed in memory and written back out when the file is closed. Keywords:

backing_store: If True (default), save changes to a real file when closing. If False, the file exists purely in memory and is discarded when 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).

4.1.3 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 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".

4.1.4 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.

4.1.5 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 `File` (*name*, *mode=None*, *driver=None*, *libver=None*, *userblock_size*, ***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 (*str* or *unicode*), or an instance of `h5f.FileID` to bind to an existing file identifier.

- **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*.
- **kwds** – Driver-specific keywords; see *File drivers*.

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. Generally a Unicode string; a byte string will be used if HDF5 returns a non-UTF-8 encoded 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.

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*.

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
u'/'
>>> f.keys()
[]
```

Names of all objects in the file are all text strings (`unicode` on Py2, `str` on Py3). 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 containership testing and iteration, but can't be sliced like lists.

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:

```
>>> f["other name"] == f["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

New in HDF5 1.8, 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.

4.2.3 Reference

class 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 containership 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*.

keys()
Get the names of directly attached group members. On Py2, this is a list. On Py3, it's a set-like object. Use *Group.visit()* or *Group.visititems()* for recursive access to group members.

values()
Get the objects contained in the group (Group and Dataset instances). Broken soft or external links show up as None. On Py2, this is a list. On Py3, it's 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. On Py2, this is a list. On Py3, it's a set-like object.

iterkeys()
(Py2 only) Get an iterator over key names. Exactly equivalent to `iter(group)`. Use *Group.visit()* or *Group.visititems()* for recursive access to group members.

itervalues()
(Py2 only) Get an iterator over objects attached to the group. Broken soft and external links will show up as `None`.

iteritems()
(Py2 only) Get an iterator over `(name, value)` pairs for objects directly attached to the group. Broken soft and external link values show up as `None`.

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)
u'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 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.

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*)
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.

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).

require_dataset (*name*, *shape=None*, *dtype=None*, *exact=None*, ***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.

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 or dtype don't match according to the above rules.

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

attrs
Attributes for this group.

id
The groups's low-level identifer; 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 HardLink
Exists only to support *Group.get()*. Has no state and provides no properties or methods.

class 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 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

path
Path to the object in the external file

4.3 Datasets

Datasets are very similar to NumPy arrays. They are homogenous 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
- **dtype** attribute

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 make an empty 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')
```

You may initialize the dataset to an existing NumPy array:

```
>>> 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 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.

4.3.3 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.

4.3.4 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.

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.

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.

4.3.5 Reading & writing data

HDF5 datasets re-use 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

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]
```

For compound data, you can specify multiple field names alongside the numeric slices:

```
>>> dset["FieldA"]
>>> dset[0,:,4:5, "FieldA", "FieldB"]
>>> dset[0, ..., "FieldC"]
```

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.

4.3.6 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:

- List selections may not be empty
- 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,)
```

4.3.7 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.

4.3.8 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 by 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 an exception if used on a empty dataset.

4.3.9 Reference

class `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*.

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])
```

astype(*dtype*)
Return a context manager 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')
>>> with dset.astype('int16'):
...     out = dset[:]
>>> out.dtype
dtype('int16')
```

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.

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.

maxshape
NumPy-style shape tuple indicating the maxiumum 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.

dims
Access to *Dimension Scales*.

attrs
Attributes for this dataset.

id
The dataset's low-level identifer; 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.

4.4.1 Reference

class `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. On Py2, this is a list. On Py3, it's a set-like object.

values()
Get the values of all attributes attached to this object. On Py2, this is a list. On Py3, it's a collection or bag-like object.

items()
Get `(name, value)` tuples for all attributes attached to this object. On Py2, this is a list of tuples. On Py3, it's a collection or set-like object.

iterkeys()
(Py2 only) Get an iterator over attribute names.

itervalues()
(Py2 only) Get an iterator over attribute values.

iteritems()
(Py2 only) Get an iterator over `(name, value)` pairs.

get(*name*, *default=None*)
Retrieve *name*, or *default* if no such attribute exists.

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 "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['data'].dims.create_scale(f['x1'])
f['data'].dims.create_scale(f['x2'], 'x2 name')
f['data'].dims.create_scale(f['y1'], 'y1 name')
f['data'].dims.create_scale(f['z1'], '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.

CHAPTER 5

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").

5.1.2 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()
```

It is also possible to configure IPython to enable the completer every time you start a new session. For >=ipython-0.11, "h5py.ipy_completer" just needs to be added 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']
```

For <ipython-0.11, the completer can be enabled by adding the following lines to the `main()` in `.ipython/ipy_user_conf.py`:

```
def main():
    ip.ex('from h5py import ipy_completer')
    ip.ex('ipy_completer.load_ipython_extension()')
```

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.

There are two functions for creating these "hinted" dtypes:

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.

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`.

5.2.2 Variable-length strings

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.special_dtype(vlen=str)
>>> ds = f.create_dataset('VLDS', (100,100), dtype=dt)
>>> ds.dtype.kind
'O'
>>> h5py.check_dtype(vlen=ds.dtype)
<type 'str'>
```

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.special_dtype(vlen=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)
```

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.special_dtype(enum=('i', {"RED": 0, "GREEN": 1, "BLUE": 42}))
>>> h5py.check_dtype(enum=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
```

5.2.5 Object and region references

References have their *own section*.

5.3 Strings in HDF5

5.3.1 The Most Important Thing

If you remember nothing else, remember this:

> **All strings in HDF5 hold encoded text.**

You *can't* store arbitrary binary data in HDF5 strings. Not only will this break, it will break in odd, hard-to-discover ways that will leave you confused and cursing.

5.3.2 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.tostring()
```

5.3.3 How to store text strings

At the high-level interface, h5py exposes three kinds of strings. Each maps to a specific type within Python (but see *Compatibility* below):

- Fixed-length ASCII (NumPy `S` type)
- Variable-length ASCII (Python 2 `str`, Python 3 `bytes`)
- Variable-length UTF-8 (Python 2 `unicode`, Python 3 `str`)

Compatibility

If you want to write maximally-compatible files and don't want to read the whole chapter:

- Use `numpy.string_` for scalar attributes
- Use the NumPy `S` dtype for datasets and array attributes

Fixed-length ASCII

These are created when you use `numpy.string_`:

```
>>> dset.attrs["name"] = numpy.string_("Hello")
```

or the `S` dtype:

```
>>> dset = f.create_dataset("string_ds", (100,), dtype="S10")
```

In the file, these map to fixed-width ASCII strings. One byte per character is used. The representation is "null-padded", which is the internal representation used by NumPy (and the only one which round-trips through HDF5).

Technically, these strings are supposed to store *only* ASCII-encoded text, although in practice anything you can store in NumPy will round-trip. But for compatibility with other progams using HDF5 (IDL, MATLAB, etc.), you should use ASCII only.

Note: This is the most-compatible way to store a string. Everything else can read it.

Variable-length ASCII

These are created when you assign a byte string to an attribute:

```
>>> dset.attrs["attr"] = b"Hello"
```

or when you create a dataset with an explicit "bytes" vlen type:

```
>>> dt = h5py.special_dtype(vlen=bytes)
>>> dset = f.create_dataset("name", (100,), dtype=dt)
```

Note that they're *not* fully identical to Python byte strings. You can only store ASCII-encoded text, without NULL bytes:

```
>>> dset.attrs["name"] = b"Hello\x00there"
ValueError: VLEN strings do not support embedded NULLs
```

In the file, these are created as variable-length strings with character set H5T_CSET_ASCII.

Variable-length UTF-8

These are created when you assign a `unicode` string to an attribute:

```
>>> dset.attrs["name"] = u"Hello"
```

or if you create a dataset with an explicit `unicode` vlen type:

```
>>> dt = h5py.special_dtype(vlen=unicode)
>>> dset = f.create_dataset("name", (100,), dtype=dt)
```

They can store any character a Python unicode string can store, with the exception of NULLs. In the file these are created as variable-length strings with character set H5T_CSET_UTF8.

Exceptions for Python 3

Most strings in the HDF5 world are stored in ASCII, which means they map to byte strings. But in Python 3, there's a strict separation between *data* and *text*, which intentionally makes it painful to handle encoded strings directly.

So, when reading or writing scalar string attributes, on Python 3 they will *always* be returned as type `str`, regardless of the underlying storage mechanism. The regular rules for writing apply; to get a fixed-width ASCII string, use `numpy.string_`, and to get a variable-length ASCII string, use `bytes`.

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 when attempting to create datasets or attributes of this type.

5.3.4 Object names

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

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

You can supply either byte or unicode strings (on both Python 2 and Python 3) when creating or retrieving objects. If a byte string is supplied, it will be used as-is; Unicode strings will be encoded down to 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(u"name2")
```

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]
```

There is one complication; since HDF5 region references don't express shapes the same way as NumPy does, the data returned will be "flattened" into a 1-D array:

```
>>> subset.shape
(50,)
```

This is similar to the behavior of NumPy's fancy indexing, which returns a 1D array for selections which don't conform to a regular grid.

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.

H5py contains a convenience function to create these "hinted dtypes" for you:

```
>>> ref_dtype = h5py.special_dtype(ref=h5py.Reference)
>>> type(ref_dtype)
<type 'numpy.dtype'>
>>> ref_dtype.kind
'O'
```

The types accepted by this "ref=" keyword argument are h5py.Reference (for object references) and h5py.RegionReference (for region references).

To create an array of references, use this dtype as you normally would:

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

Starting with version 2.2.0, h5py includes support for Parallel HDF5. This is the "native" way to use HDF5 in a parallel computing environment.

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)
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". This is simple; set your default compiler to the `mpicc` wrapper and build h5py with the `--mpi` option:

```
$ export CC=mpicc
$ python setup.py configure --mpi [--hdf5=/path/to/parallel/hdf5]
$ python setup.py build
```

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 versions 1.8.9+ support 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.

The SWMR feature is not available in the current release (1.8 series) of HDF5 library. It is planned to be released for production use in version 1.10. Until then it is available as an experimental prototype form from development snapshot version 1.9.178 on the HDF Group ftp server or the HDF Group svn repository.

Warning: The SWMR feature is currently in prototype form and available for experimenting and testing. Please do not consider this a production quality feature until the HDF5 library is released as 1.10.

Warning: FILES PRODUCED BY THE HDF5 1.9.X DEVELOPMENT SNAPSHOTS MAY NOT BE READABLE BY OTHER VERSIONS OF HDF5, INCLUDING THE EXISTING 1.8 SERIES AND ALSO 1.10 WHEN IT IS RELEASED.

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 syncrhonised 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 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 create the target file and all groups, datasets and attributes.
- Writer process switch 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 demonstrate 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 nofication 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.
"""
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"
    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 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, time
import h5py
import numpy as np
import logging
from multiprocessing import Process, Event

class SwmrReader(Process):
    def __init__(self, event, fname, dsetname, timeout = 2.0):
        super(SwmrReader, self).__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(SwmrWriter, self).__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")
            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
    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
  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
```

CHAPTER 6

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 2.7

Python 3.2 is no longer supported

`h5py` 2.7 drops Python 3.2 support, and testing is not longer preformed 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 #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 (#795, #798, #799 and #801 by James Tocknell) and testing of minimum requirements to ensure we still satisfy them (#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 (#662, #700 and #733). Thanks to to the Appveyor support, unicode tests were made more robust (#788, #800 and #804 by James Tocknell). Finally, other tests were improved or added where needed (#724 by Matthew Brett, #789, #794 and #802 by James Tocknell).

Improved python compatibility

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

Documentation improvements

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

setup.py improvements

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

Support for additional HDF5 features added

Low-level support for HDF5 Direct Chunk Write was added in #691 by Simon Gregor Ebner. Minimal support for HDF5 File Image Operations was added by Andrea Bedini in #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 #664 by James Tocknell.

Improvements to type system

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

Other changes

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

Acknowlegements

6.1.2 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 #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 #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 #576 by Jakob Lombacher, #640 by Lawrence Mitchell, and #650, #651 and #658 by James Tocknell.

Cleanup of codebase based on pylint

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

Fixes to low-level API

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

Documentation improvements

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

Other changes

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

Acknowlegements

6.1.3 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: http://docs.h5py.org/en/latest/swmr.html

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

Other changes

- Use system Cython as a fallback if *cythonize()* fails (#541 by Ulrik Pedersen).
- Use pkg-config for builing/linking against hdf5 (#505 by James Tocknell).
- Disable building Cython on Travis (#513 by Andrew Collette).
- Improvements to release tarball (#555, #560 by Ghislain Antony Vaillant).
- h5py now has one codebase for both Python 2 and 3; 2to3 removed from setup.py (#508 by James Tocknell).
- Add python 3.4 to tox (#507 by James Tocknell).
- Warn when importing from inside install dir (#558 by Andrew Collette).
- Tweak installation docs with reference to Anaconda and other Python package managers (#546 by Andrew Collette).
- Fix incompatible function pointer types (#526, #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 (#527 by James Tocknell).
- Don't strip metadata from special dtypes on read (#512 by Antony Lee).
- Add 'x' mode as an alias for 'w-' (#510 by Antony Lee).
- Support dynamical loading of LZF filter plugin (#506 by Peter Colberg).
- Fix accessing attributes with array type (#501 by Andrew Collette).
- Don't leak types in enum converter (#503 by Andrew Collette).

Acknowlegements

6.1.4 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

6.1.5 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: 10

IOError: Unable to open file (Unable to open file: name = 'x3', errno = 2, error message = 'no such f

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

6.1.6 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.

6.1.7 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)

6.1.8 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 extendible 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, always feel free to ask on the Google Groups list, either by emailing:

h5py@googlegroups.com

or via the web interface at:

https://groups.google.com/forum/#!forum/h5py

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:

http://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

If you're fixing a bug, you'll want to check out a branch against the appropriate stable branch. For example, to fix a bug you found in version 2.1.3, you'll want to check out against branch "2.1":

```
$ git checkout -b bugfix 2.1
```

If you're contributing a new feature, it's appropriate to develop against the "master" branch, so you would instead do:

```
$ 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.

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.

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 whether the project builds correctly, using Travis CI. Check to see if the build succeeded (takes about 5 minutes), 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):
        """ 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 backwards-compatible all the way back to HDF5 1.8.4. Starting with h5py 2.2.0, it's possible to conditionally include functions which only appear in newer versions of HDF5. It's also possible to mark functions which requre 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.

6.3 FAQ

6.3.1 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
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*

Unsupported types:

Type	Status
HDF5 "time" type	
NumPy "U" strings	No HDF5 equivalent
NumPy generic "O"	Not planned

6.3.2 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

6.3.3 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	

6.3.4 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.

6.3.5 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.

6.3.6 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.

6.3.7 Enumerated types

HDF5 enumerated types are supported as. 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.

6.3.8 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.

6.3.9 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

6.3.10 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.

6.3.11 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.

6.4 Licenses and legal info

6.4.1 Copyright Notice and Statement for the h5py Project

```
Copyright (c) 2008 Andrew Collette and contributors
http://h5py.alfven.org
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 author nor the names of 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.
```

6.4.2 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.
   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.
```

6.4.3 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.
```

6.4.4 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
     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.
```

6.4.5 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.

Symbols

A

C

D

E

F

G

H

I

K

L

M

N

P

R

S

U

V