O'REILLY®

Python and HDF5

UNLOCKING SCIENTIFIC DATA

Andrew Collette

Python and HDF5

Andrew Collette

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Special Upgrade Offer

If you purchased this ebook directly from oreilly.com, you have the following benefits:

- DRM-free ebooks—use your ebooks across devices without restrictions or limitations
- Multiple formats—use on your laptop, tablet, or phone
- Lifetime access, with free updates
- Dropbox syncing—your files, anywhere

If you purchased this ebook from another retailer, you can upgrade your ebook to take advantage of a
these benefits for just $4.99. Click here to access your ebook upgrade.

Please note that upgrade offers are not available from sample content.

Preface

Over the past several years, Python has emerged as a credible alternative to scientific analysis environments like IDL or MATLAB. Stable core packages now exist for handling numerical arrays (NumPy), analysis (SciPy), and plotting (matplotlib). A huge selection of more specialized software also available, reducing the amount of work necessary to write scientific code while also increasing the quality of results.

As Python is increasingly used to handle large numerical datasets, more emphasis has been placed or the use of standard formats for data storage and communication. HDF5, the most recent version of th "Hierarchical Data Format" originally developed at the National Center for Supercomputing Applications (NCSA), has rapidly emerged as the mechanism of choice for storing scientific data in Python. At the same time, many researchers who use (or are interested in using) HDF5 have been drawn to Python for its ease of use and rapid development capabilities.

This book provides an introduction to using HDF5 from Python, and is designed to be useful to anyor with a basic background in Python data analysis. Only familiarity with Python and NumPy is assume Special emphasis is placed on the native HDF5 feature set, rather than higher-level abstractions on th Python side, to make the book as useful as possible for creating portable files.

Finally, this book is intended to support both users of Python 2 and Python 3. While the examples are written for Python 2, any differences that may trip you up are noted in the text.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This icon signifies a tip, suggestion, or general note.

WARNING

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of examp code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Python and HDF5* by Andrew Collette (O'Reilly). Copyright 201 Andrew Collette, 978-1-449-36783-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free t contact us at permissions@oreilly.com.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training video and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Pres Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us onlin

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://oreil.ly/python-HDF5.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

I would like to thank Quincey Koziol, Elena Pourmal, Gerd Heber, and the others at the HDF Group for supporting the use of HDF5 by the Python community. This book benefited greatly from reviewer comments, including those by Eli Bressert and Anthony Scopatz, as well as the dedication and guidance of O’Reilly editor Meghan Blanchette.

Darren Dale and many others deserve thanks for contributing to the h5py project, along with Frances Alted, Antonio Valentino, and fellow authors of PyTables who first brought the HDF5 and Python worlds together. I would also like to thank Steve Vincena and Walter Gekelman of the UCLA Basic Plasma Science Facility, where I first began working with large-scale scientific datasets.

Chapter 1. Introduction

When I was a graduate student, I had a serious problem: a brand-new dataset, made up of millions of data points collected painstakingly over a full week on a nationally recognized plasma research device, that contained values that were much too small.

About *40 orders of magnitude* too small.

My advisor and I huddled in his office, in front of the shiny new G5 Power Mac that ran our visualization suite, and tried to figure out what was wrong. The data had been acquired correctly from the machine. It looked like the original raw file from the experiment's digitizer was fine. I had written a (very large) script in the IDL programming language on my Thinkpad laptop to turn the raw data into files the visualization tool could use. This in-house format was simplicity itself: just a short fixed-width header and then a binary dump of the floating-point data. Even so, I spent another hour or so writing a program to verify and plot the files on my laptop. They were fine. And yet, when loaded into the visualizer, all the data that looked so beautiful in IDL turned into a featureless, unstructured mush of values all around 10^{-41}.

Finally it came to us: both the digitizer machines and my Thinkpad used the "little-endian" format to represent floating-point numbers, in contrast to the "big-endian" format of the G5 Mac. Raw values written on one machine couldn't be read on the other, and vice versa. I remember thinking *that's so stupid* (among other less polite variations). Learning that this problem was so common that IDL supplied a special routine to deal with it (`SWAP_ENDIAN`) did not improve my mood.

At the time, I didn't care that much about the details of how my data was stored. This incident and others like it changed my mind. As a scientist, I eventually came to recognize that the choices we make for organizing and storing our data are also choices about communication. Not only do standard, well-designed formats make life easier for individuals (and eliminate silly time-wasters like the "endian" problem), but they make it possible to share data with a global audience.

Python and HDF5

In the Python world, consensus is rapidly converging on Hierarchical Data Format version 5, or "HDF5," as the standard mechanism for storing large quantities of numerical data. As data volumes get larger, organization of data becomes increasingly important; features in HDF5 like named datasets (Chapter 3), hierarchically organized groups (Chapter 5), and user-defined metadata "attributes" (Chapter 6) become essential to the analysis process.

Structured, "self-describing" formats like HDF5 are a natural complement to Python. Two production-ready, feature-rich interface packages exist for HDF5, h5py, and PyTables, along with a number of smaller special-purpose wrappers.

Organizing Data and Metadata

Here's a simple example of how HDF5's structuring capability can help an application. Don't worry too much about the details; later chapters explain both the details of how the file is structured, and how to use the HDF5 API from Python. Consider this a taste of what HDF5 can do for your application. If you want to follow along, you'll need Python 2 with NumPy installed (see Chapter 2).

Suppose we have a NumPy array that represents some data from an experiment:

```
>>> import numpy as np
>>> temperature = np.random.random(1024)
>>> temperature
array([ 0.44149738,  0.7407523 ,  0.44243584, ...,  0.19018119,
        0.64844851,  0.55660748])
```

Let's also imagine that these data points were recorded from a weather station that sampled the temperature, say, every 10 seconds. In order to make sense of the data, we have to record that sampling interval, or "delta-T," somewhere. For now we'll put it in a Python variable:

```
>>> dt = 10.0
```

The data acquisition started at a particular time, which we will also need to record. And of course, we have to know that the data came from Weather Station 15:

```
>>> start_time = 1375204299  # in Unix time
>>> station = 15
```

We could use the built-in NumPy function `np.savez` to store these values on disk. This simple function saves the values as NumPy arrays, packed together in a ZIP file with associated names:

```
>>> np.savez("weather.npz", data=temperature, start_time=start_time, station=
station)
```

We can get the values back from the file with `np.load`:

```
>>> out = np.load("weather.npz")
>>> out["data"]
array([ 0.44149738,  0.7407523 ,  0.44243584, ...,  0.19018119,
        0.64844851,  0.55660748])
>>> out["start_time"]
array(1375204299)
>>> out["station"]
array(15)
```

So far so good. But what if we have more than one quantity per station? Say there's also wind speed data to record?

```
>>> wind = np.random.random(2048)
>>> dt_wind = 5.0   # Wind sampled every 5 seconds
```

And suppose we have multiple stations. We could introduce some kind of naming convention, I suppose: "wind_15" for the wind values from station 15, and things like "dt_wind_15" for the sampling interval. Or we could use multiple files…

In contrast, here's how this application might approach storage with HDF5:

```
>>> import h5py
>>> f = h5py.File("weather.hdf5")
>>> f["/15/temperature"] = temperature
>>> f["/15/temperature"].attrs["dt"] = 10.0
>>> f["/15/temperature"].attrs["start_time"] = 1375204299
>>> f["/15/wind"] = wind
>>> f["/15/wind"].attrs["dt"] = 5.0
---
>>> f["/20/temperature"] = temperature_from_station_20
---

(and so on)
```

This example illustrates two of the "killer features" of HDF5: organization in hierarchical groups and attributes. Groups, like folders in a filesystem, let you store related datasets together. In this case, temperature and wind measurements from the same weather station are stored together under groups named "/15," "/20," etc. Attributes let you attach descriptive metadata *directly to the data it describe* So if you give this file to a colleague, she can easily discover the information needed to make sense o the data:

```
>>> dataset = f["/15/temperature"]
>>> for key, value in dataset.attrs.iteritems():
...     print "%s: %s" % (key, value)
dt: 10.0
start_time: 1375204299
```

Coping with Large Data Volumes

As a high-level "glue" language, Python is increasingly being used for rapid visualization of big datasets and to coordinate large-scale computations that run in compiled languages like C and FORTRAN. It's now relatively common to deal with datasets hundreds of gigabytes or even terabytes in size; HDF5 itself can scale up to exabytes.

On all but the biggest machines, it's not feasible to load such datasets directly into memory. One of HDF5's greatest strengths is its support for subsetting and partial I/O. For example, let's take the 1024-element "temperature" dataset we created earlier:

```
>>> dataset = f["/15/temperature"]
```

Here, the object named `dataset` is a proxy object representing an HDF5 dataset. It supports array-lik slicing operations, which will be familiar to frequent NumPy users:

```
>>> dataset[0:10]
```

```
array([ 0.44149738,  0.7407523 ,  0.44243584,  0.3100173 ,  0.04552416,
        0.43933469,  0.28550775,  0.76152561,  0.79451732,  0.32603454])
>>> dataset[0:10:2]
array([ 0.44149738,  0.44243584,  0.04552416,  0.28550775,  0.79451732])
```

Keep in mind that the actual data lives on disk; when slicing is applied to an HDF5 dataset, the appropriate data is found and loaded into memory. Slicing in this fashion leverages the underlying subsetting capabilities of HDF5 and is consequently very fast.

Another great thing about HDF5 is that you have control over how storage is allocated. For example, except for some metadata, a brand new dataset takes *zero* space, and by default bytes are only used o disk to hold the data you actually write.

For example, here's a 2-terabyte dataset you can create on just about any computer:

```
>>> big_dataset = f.create_dataset("big", shape=(1024, 1024, 1024, 512), dtype='float32')
```

Although no storage is yet allocated, the entire "space" of the dataset is available to us. We can write anywhere in the dataset, and only the bytes on disk necessary to hold the data are used:

```
>>> big_dataset[344, 678, 23, 36] = 42.0
```

When storage is at a premium, you can even use transparent compression on a dataset-by-dataset bas (see Chapter 4):

```
>>> compressed_dataset = f.create_dataset("comp", shape=(1024,), dtype='int32', compression='gzip')
>>> compressed_dataset[:] = np.arange(1024)
>>> compressed_dataset[:]
array([   0,    1,    2, ..., 1021, 1022, 1023])
```

What Exactly Is HDF5?

HDF5 is a great mechanism for storing *large numerical arrays of homogenous type*, for data models that can be *organized hierarchically* and benefit from tagging of datasets with *arbitrary metadata*.

It's quite different from SQL-style relational databases. HDF5 has quite a few organizational tricks u its sleeve (see Chapter 8, for example), but if you find yourself needing to enforce relationships between values in various tables, or wanting to perform JOINs on your data, a relational database is probably more appropriate. Likewise, for tiny 1D datasets you need to be able to read on machines without HDF5 installed. Text formats like CSV (with all their warts) are a reasonable alternative.

HDF5 is just about perfect if you make minimal use of relational features and have a need for very high performance, partial I/O, hierarchical organization, and arbitrary metadata.

So what, specifically, is "HDF5"? I would argue it consists of three things:

1. A file specification and associated data model.

2. A standard library with API access available from C, C++, Java, Python, and others.

3. A software ecosystem, consisting of both client programs using HDF5 and "analysis platforms" like MATLAB, IDL, and Python.

HDF5: The File

In the preceding brief examples, you saw the three main elements of the HDF5 data model: *datasets*, array-like objects that store your numerical data on disk; *groups*, hierarchical containers that store datasets and other groups; and *attributes*, user-defined bits of metadata that can be attached to datase (and groups!).

Using these basic abstractions, users can build specific "application formats" that organize data in a method appropriate for the problem domain. For example, our "weather station" code used one group for each station, and separate datasets for each measured parameter, with attributes to hold additiona information about what the datasets mean. It's very common for laboratories or other organizations t agree on such a "format-within-a-format" that specifies what arrangement of groups, datasets, and attributes are to be used to store information.

Since HDF5 takes care of all cross-platform issues like endianness, sharing data with other groups becomes a simple matter of manipulating groups, datasets, and attributes to get the desired result. An because the files are *self-describing*, even knowing about the application format isn't usually necessary to get data out of the file. You can simply open the file and explore its contents:

```
>>> f.keys()
[u'15', u'big', u'comp']
>>> f["/15"].keys()
[u'temperature', u'wind']
```

Anyone who has spent hours fiddling with byte-offsets while trying to read "simple" binary formats can appreciate this.

Finally, the low-level byte layout of an HDF5 file on disk is an open specification. There are no mysteries about how it works, in contrast to proprietary binary formats. And although people typicall use the library provided by the HDF Group to access files, nothing prevents you from writing your own reader if you want.

HDF5: The Library

The HDF5 file specification and open source library is maintained by the HDF Group, a nonprofit organization headquartered in Champaign, Illinois. Formerly part of the University of Illinois Urbana Champaign, the HDF Group's primary product is the HDF5 software library.

Written in C, with additional bindings for C++ and Java, this library is what people usually mean when they say "HDF5." Both of the most popular Python interfaces, PyTables and h5py, are designed to use the C library provided by the HDF Group.

One important point to make is that this library is actively maintained, and the developers place a strong emphasis on backwards compatibility. This applies to both the files the library produces and also to programs that use the API. File compatibility is a must for an archival format like HDF5. Suc

careful attention to API compatibility is the main reason that packages like h5py and PyTables have been able to get traction with many different versions of HDF5 installed in the wild.

You should have confidence when using HDF5 for scientific data storage, including long-term storag And since both the library and format are open source, your files will be readable even if a meteor takes out Illinois.

HDF5: The Ecosystem

Finally, one aspect that makes HDF5 particularly useful is that you can read and write files from just about every platform. The IDL language has supported HDF5 for years; MATLAB has similar suppo and now even uses HDF5 as the default format for its ".mat" save files. Bindings are also available fo Python, C++, Java, .NET, and LabView, among others. Institutional users include NASA's Earth Observing System, whose "EOS5" format is an application format on top of the HDF5 container, as i the much simpler example earlier. Even the newest version of the competing NetCDF format, NetCDF4, is implemented using HDF5 groups, datasets, and attributes.

Hopefully I've been able to share with you some of the things that make HDF5 so exciting for scientific use. Next, we'll review the basics of how HDF5 works and get started on using it from Python.

Chapter 2. Getting Started

HDF5 Basics

Before we jump into Python code examples, it's useful to take a few minutes to address how HDF5 itself is organized. Figure 2-1 shows a cartoon of the various logical layers involved when using HDF5. Layers shaded in blue are internal to the library itself; layers in green represent software that uses HDF5.

Most client code, including the Python packages h5py and PyTables, uses the native *C API* (HDF5 is itself written in C). As we saw in the introduction, the HDF5 data model consists of three main *public abstractions*: datasets (see Chapter 3), groups (see Chapter 5), and attributes (see Chapter 6)in addition to a system to represent types. The C API (and Python code on top of it) is designed to manipulate these objects.

HDF5 uses a variety of *internal data structures* to represent groups, datasets, and attributes. For example, groups have their entries indexed using structures called "B-trees," which make retrieving and creating group members very fast, even when hundreds of thousands of objects are stored in a group (see How Groups Are Actually Stored). You'll generally only care about these data structures when it comes to performance considerations. For example, when using chunked storage (see Chapter 4), it's important to understand how data is actually organized on disk.

The next two layers have to do with how your data makes its way onto disk. HDF5 objects all live in 1D logical address space, like in a regular file. However, there's an extra layer between this space an the actual arrangement of bytes on disk. HDF5 *drivers* take care of the mechanics of writing to disk, and in the process can do some amazing things.

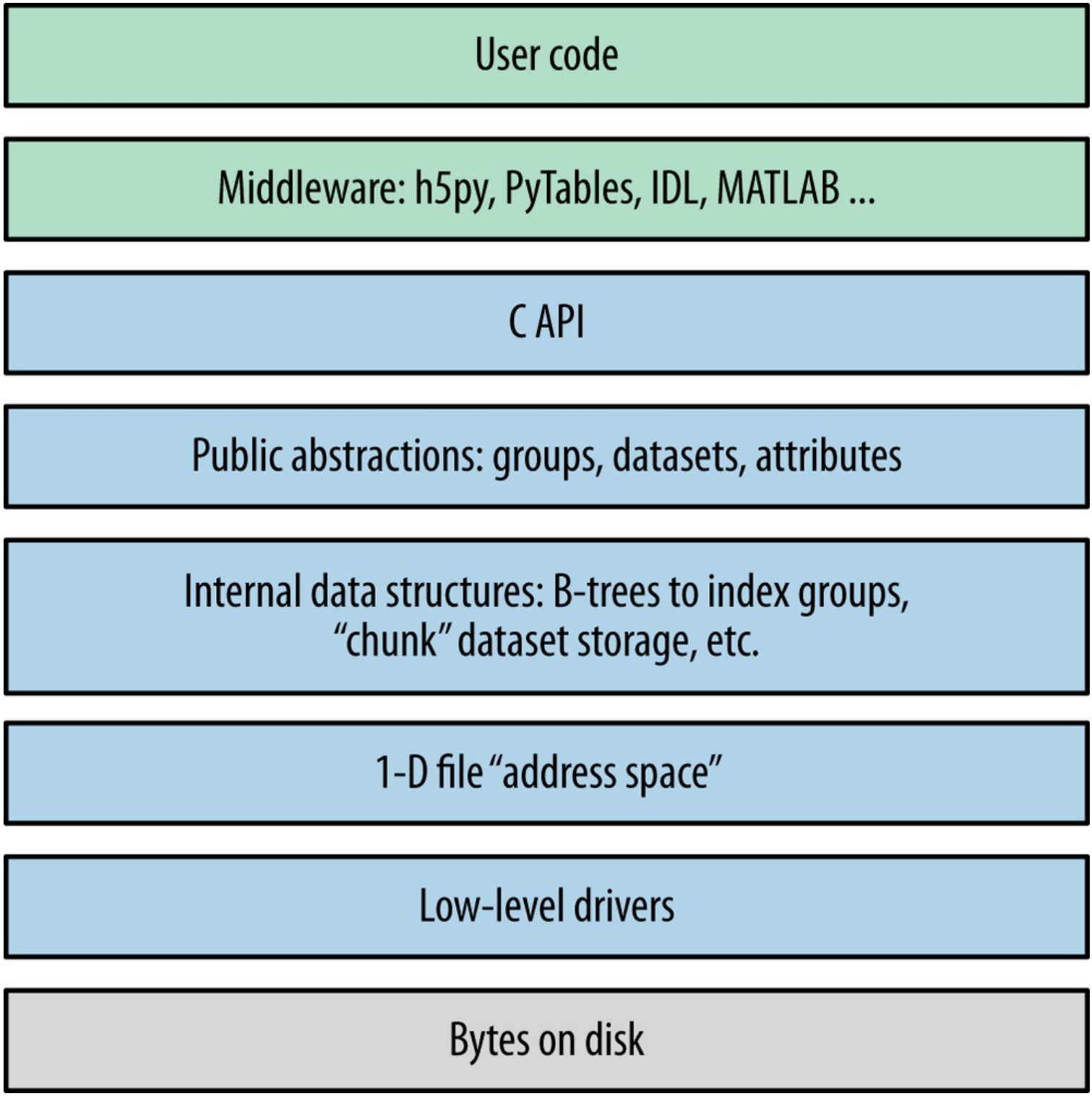

Figure 2-1. The HDF5 library: blue represents components inside the HDF5 library; green represents "client" code that calls into HDF5; gray represents resources provided by the operating system.

For example, the HDF5 `core` driver lets you use files that live entirely in memory and are blazingly fast. The `family` driver lets you split a single file into regularly sized pieces. And the `mpio` driver let you access the same file from multiple parallel processes, using the Message Passing Interface (MPI library (MPI and Parallel HDF5). All of this is transparent to code that works at the higher level of groups, datasets, and attributes.

Setting Up

That's enough background material for now. Let's get started with Python! But *which* Python?

Python 2 or Python 3?

A big shift is under way in the Python community. Over the years, Python has accumulated a number of features and misfeatures that have been deemed undesirable. Examples range from packages that use inconsistent naming conventions all the way to deficiencies in how strings are handled. To addres these issues, it was decided to launch a new major version (Python 3) that would be freed from the "baggage" of old decisions in the Python 2 line.

Python 2.7, the most recent minor release in the Python series, will also be the *last* 2.X release. Although it will be updated with bug fixes for an extended period of time, new Python code development is now carried out exclusively on the 3.X line. The NumPy package, h5py, PyTables, an many other packages now support Python 3. While (in my opinion) it's a little early to recommend that newcomers start with Python 3, the future is clear.

So at the moment, there are two major versions of Python widely deployed simultaneously. Since most people in the Python community are used to Python 2, the examples in this book are also writte for Python 2. For the most part, the differences are minor; for example, on Python 3, the syntax for `print` is `print(foo)`, instead of `print foo`. Wherever incompatibilities are likely to occur (mainly with string types and certain dictionary-style methods), these will be noted in the text.

"Porting" code to Python 3 isn't actually that hard; after all, it's still Python. Some of the most valuable features in Python 3 are already present in Python 2.7. A free tool is also available (`2to3`) that can accomplish most of the mechanical changes, for example changing `print` statements to `print()` function calls. Check out the migration guide (and the `2to3` tool) at http://www.python.org.

Code Examples

To start with, most of the code examples will follow this form:

```
>>> a = 1.0
>>> print a
1.0
```

Or, since Python prints objects by default, an even simpler form:

```
>>> a
1.0
```

Lines starting with `>>>` represent input to Python (`>>>` being the default Python prompt); other lines represent output. Some of the longer examples, where the program output is not shown, omit the `>>>` in the interest of clarity.

Examples intended to be run from the command line will start with the Unix-style "$" prompt:

```
$ python --version
Python 2.7.3
```

Finally, to avoid cluttering up the examples, most of the code snippets you'll find here will assume that the following packages have been imported:

```
>>> import numpy as np      # Provides array object and data type objects
>>> import h5py             # Provides access to HDF5
```

NumPy

"NumPy" is the standard numerical-array package in the Python world. This book assumes that you have some experience with NumPy (and of course Python itself), including how to use array objects.

Even if you've used NumPy before, it's worth reviewing a few basic facts about how arrays work. First, NumPy arrays all have a fixed data type or "dtype," represented by *dtype objects*. For example, let's create a simple 10-element integer array:

```
>>> arr = np.arange(10)
>>> arr
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The data type of this array is given by `arr.dtype`:

```
>>> arr.dtype
dtype('int32')
```

These dtype objects are how NumPy communicates low-level information about the type of data in memory. For the case of our 10-element array of 4-byte integers (32-bit or "int32" in NumPy lingo), there is a chunk of memory somewhere 40 bytes long that holds the values 0 through 9. Other code that receives the `arr` object can inspect the dtype object to figure out what the memory contents represent.

> **NOTE**
>
> The preceding example might print `dtype('int64')` on your system. All this means is that the default integer size available to Python is 64 bits long, instead of 32 bits.

HDF5 uses a very similar type system; every "array" or *dataset* in an HDF5 file has a fixed type represented by a type object. The `h5py` package automatically maps the HDF5 type system onto NumPy dtypes, which among other things makes it easy to interchange data with NumPy. Chapter 3 goes into more detail about this process.

Slicing is another core feature of NumPy. This means accessing *portions* of a NumPy array. For example, to extract the first four elements of our array `arr`:

```
>>> out = arr[0:4]
>>> out
array([0, 1, 2, 3])
```

You can also specify a “stride” or steps between points in the slice:

```
>>> out = arr[0:4:2]
>>> out
array([0, 2])
```

When talking to HDF5, we will borrow this “slicing” syntax to allow loading only portions of a dataset.

In the NumPy world, slicing is implemented in a clever way, which generally creates arrays that *refe* to the data in the original array, rather than independent copies. For example, the preceding `out` obje is likely a “view” onto the original `arr` array. We can test this:

```
>>> out[:] = 42
>>> out
array([42, 42])
>>> arr
array([42,  1, 42,  3,  4,  5,  6,  7,  8,  9])
```

This means slicing is generally a very fast operation. But you should be careful to explicitly create a copy if you want to modify the resulting array without your changes finding their way back to the original:

```
>>> out2 = arr[0:4:2].copy()
```

> **CAUTION**
>
> Forgetting to make a copy before modifying a “slice” of the array is a very common mistake, especially for people used to environments like IDL. If you’re new to NumPy, be careful!

As we’ll see later, thankfully this doesn’t apply to slices read from HDF5 datasets. When you read from the file, since the data is on disk, you will always get a copy.

HDF5 and h5py

We’ll use the “h5py” package to talk to HDF5. This package contains high-level wrappers for the HDF5 objects of files, groups, datasets, and attributes, as well as extensive low-level wrappers for th HDF5 C data structures and functions. The examples in this book assume h5py version 2.2.0 or later, which you can get at http://www.h5py.org.

You should note that h5py is not the only commonly used package for talking to HDF5. PyTables is a scientific database package based on HDF5 that adds dataset indexing and an additional type system. Since we’ll be talking about native HDF5 constructs, we’ll stick to h5py, but I strongly recommend

you also check out PyTables if those features interest you.

If you're on Linux, it's generally a good idea to install the HDF5 library via your package manager. On Windows, you can download an installer from http://www.h5py.org, or use one of the many distributions of Python that include HDF5/h5py, such as PythonXY, Anaconda from Continuum Analytics, or Enthought Python Distribution.

IPython

Apart from NumPy and h5py/HDF5 itself, IPython is a must-have component if you'll be doing extensive analysis or development with Python. At its most basic, IPython is a replacement interpret shell that adds features like command history and Tab-completion for object attributes. It also has tons of additional features for parallel processing, MATLAB-style "notebook" display of graphs, and more.

The best way to explore the features in this book is with an IPython prompt open, following along wi the examples. Tab-completion alone is worth it, because it lets you quickly see the attributes of modules and objects. The h5py package is specifically designed to be "explorable" in this sense. For example, if you want to discover what properties and methods exist on the `File` object (see Your Firs HDF5 File), type `h5py.File.` and bang the Tab key:

```
>>> h5py.File.<TAB>
h5py.File.attrs            h5py.File.get             h5py.File.name
h5py.File.close            h5py.File.id              h5py.File.parent
h5py.File.copy             h5py.File.items           h5py.File.ref
h5py.File.create_dataset   h5py.File.iteritems       h5py.File.require_dataset
h5py.File.create_group     h5py.File.iterkeys        h5py.File.require_group
h5py.File.driver           h5py.File.itervalues      h5py.File.userblock_size
h5py.File.fid              h5py.File.keys            h5py.File.values
h5py.File.file             h5py.File.libver          h5py.File.visit
h5py.File.filename         h5py.File.mode            h5py.File.visititems
h5py.File.flush            h5py.File.mro
```

To get more information on a property or method, use `?` after its name:

```
>>> h5py.File.close?
Type:       instancemethod
Base Class: <type 'instancemethod'>
String Form:<unbound method File.close>
Namespace:  Interactive
File:       /usr/local/lib/python2.7/dist-packages/h5py/_hl/files.py
Definition: h5py.File.close(self)
Docstring:  Close the file.  All open objects become invalid
```

NOTE

By default, IPython will save the output of your statements in special hidden variables. This is generally OK, but can be surprising if it hangs on to an HDF5 object you thought was discarded, or a big array that eats up memory. You can turn this off by setting the IPython configuration value `cache_size` to 0. See the docs at http://ipython.org for more information.

Timing and Optimization

For performance testing, we'll use the `timeit` module that ships with Python. Examples using `timei` will assume the following import:

```
>>> from timeit import timeit
```

The `timeit` function takes a (string or callable) command to execute, and an optional number of time it should be run. It then prints the total time spent running the command. For example, if we execute the "wait" function `time.sleep` five times:

```
>>> import time
>>> timeit("time.sleep(0.1)", number=5)
0.49967818316434887
```

If you're using IPython, there's a similar built-in "magic" function called `%timeit` that runs the specified statement a few times, and reports the lowest execution time:

```
>>> %timeit time.sleep(0.1)
10 loops, best of 3: 100 ms per loop
```

We'll stick with the regular `timeit` function in this book, in part because it's provided by the Python standard library.

Since people using HDF5 generally deal with large datasets, performance is always a concern. But you'll notice that optimization and benchmarking discussions in this book don't go into great detail about things like cache hits, data conversion rates, and so forth. The design of the h5py package, which this book uses, leaves nearly all of that to HDF5. This way, you benefit from the hundreds of man years of work spent on tuning HDF5 to provide the highest performance possible.

As an application builder, the best thing you can do for performance is to use the API in a sensible way and let HDF5 do its job. Here are some suggestions:

1. Don't optimize anything unless there's a demonstrated performance problem. Then, carefully isolate the misbehaving parts of the code before changing anything.

2. Start with simple, straightforward code that takes advantage of the API features. For example, t iterate over all objects in a file, use the Visitor feature of HDF5 (see Multilevel Iteration with the Visitor Pattern) rather than cobbling together your own approach.

3. Do "algorithmic" improvements first. For example, when writing to a dataset (see Chapter 3),

write data in reasonably sized blocks instead of point by point. This lets HDF5 use the filesyste
in an intelligent way.

4. Make sure you're using the right data types. For example, if you're running a compute-intensiv
program that loads floating-point data from a file, make sure that you're not accidentally using
double-precision floats in a calculation where single precision would do.

5. Finally, don't hesitate to ask for help on the h5py or NumPy/Scipy mailing lists, Stack Overflo
or other community sites. Lots of people are using NumPy and HDF5 these days, and many
performance problems have known solutions. The Python community is very welcoming.

The HDF5 Tools

We'll be creating a number of files in later chapters, and it's nice to have an independent way of seeing what they contain. It's also a good idea to inspect files you create professionally, especially if you'll be using them for archiving or sharing them with colleagues. The earlier you can detect the us of an incorrect data type, for example, the better off you and other users will be.

HDFView

HDFView is a free graphical browser for HDF5 files provided by the HDF Group. It's a little basic, but is written in Java and therefore available on Windows, Linux, and Mac. There's a built-in spreadsheet-like browser for data, and basic plotting capabilities.

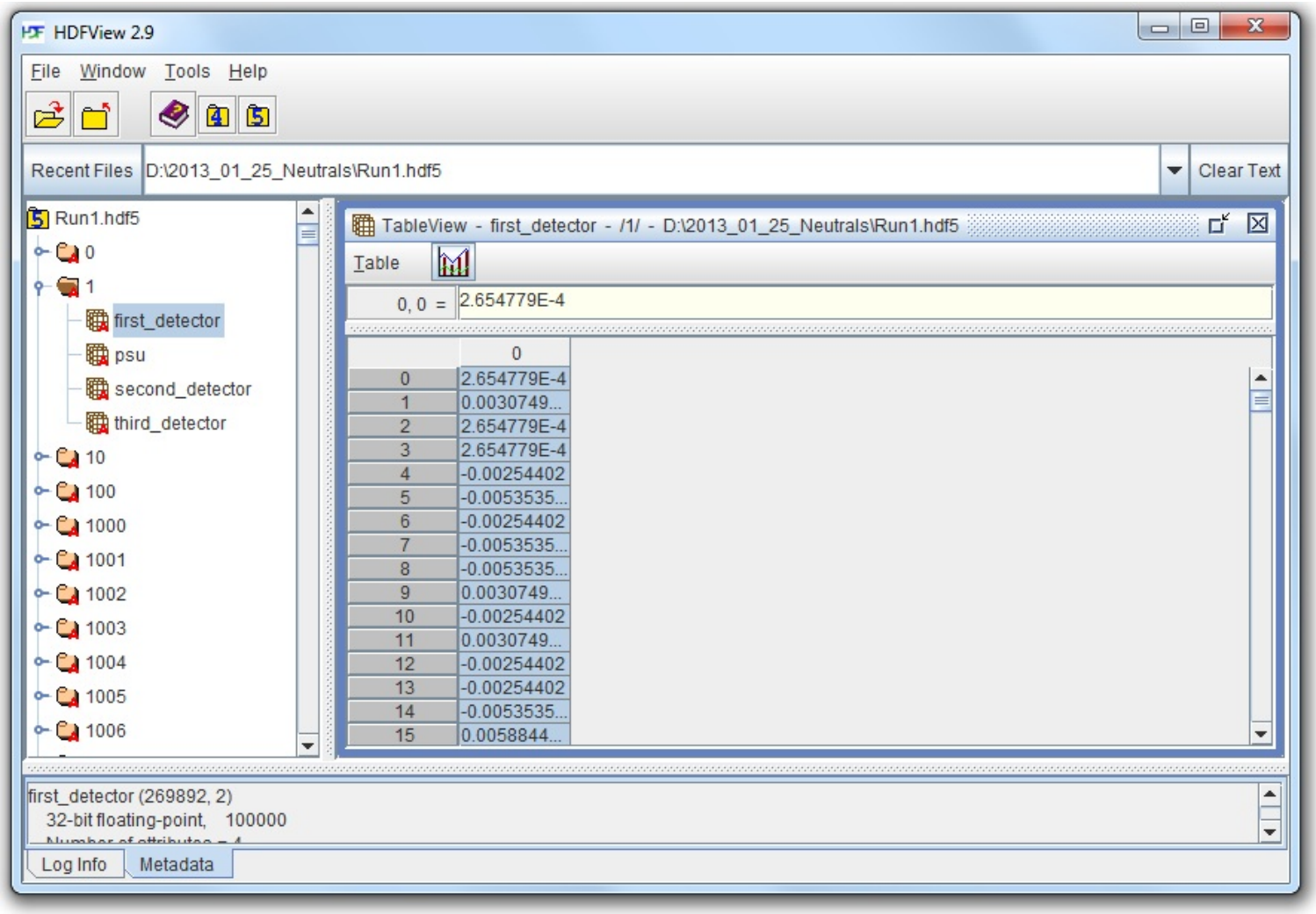

Figure 2-2. HDFView

Figure 2-2 shows the contents of an HDF5 file with multiple groups in the left-hand pane. One group (named “1”) is open, showing the datasets it contains; likewise, one dataset is opened, with its contents displayed in the grid view to the right.

HDFView also lets you inspect attributes of datasets and groups, and supports nearly all the data type that HDF5 itself supports, with the exception of certain variable-length structures.

ViTables

Figure 2-3 shows the same HDF5 file open in ViTables, another free graphical browser. It’s optimize for dealing with PyTables files, although it can handle generic HDF5 files perfectly well. One major advantage of ViTables is that it comes preinstalled with such Python distributions as PythonXY, so you may already have it.

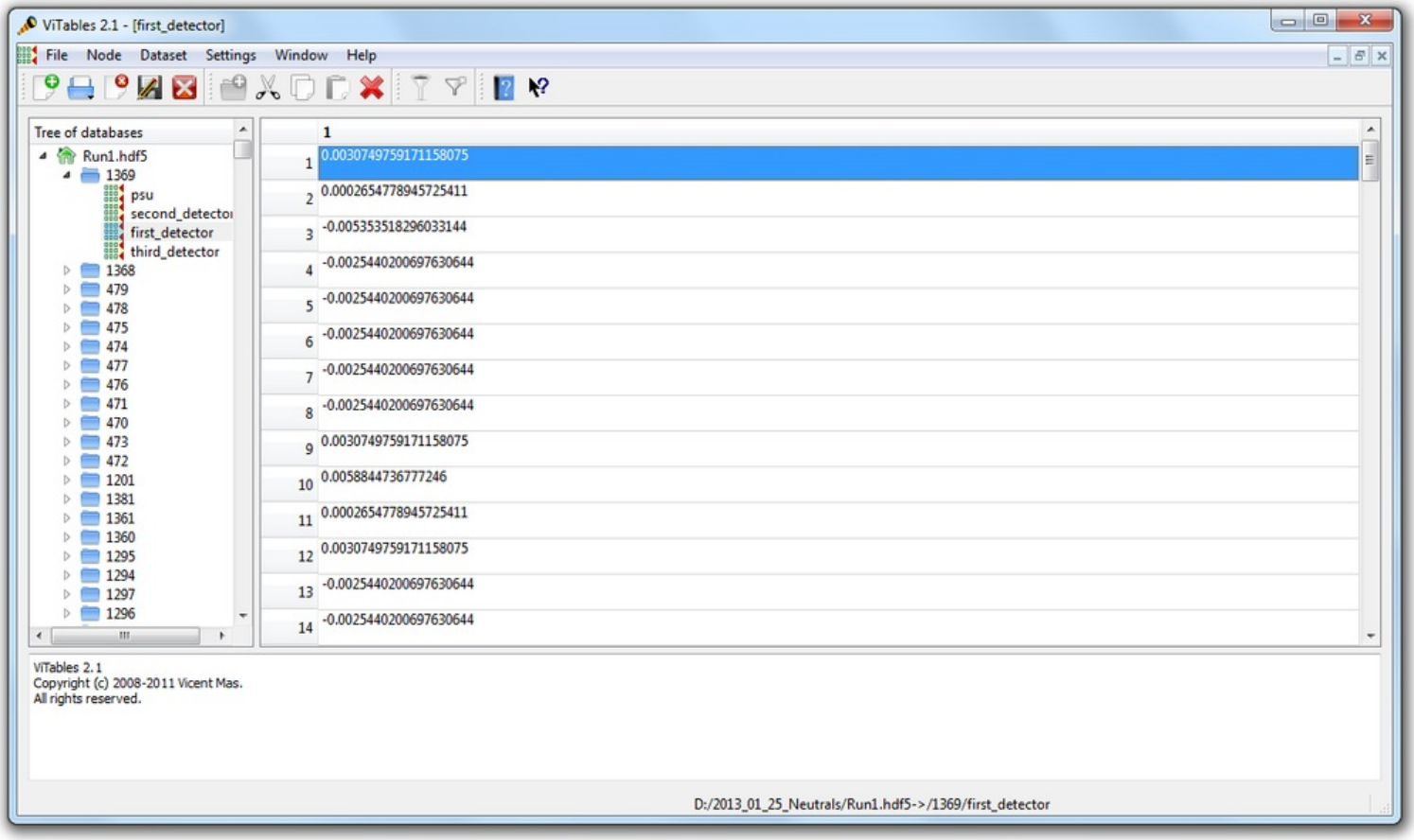

Figure 2-3. ViTables

Command Line Tools

If you're used to the command line, it's definitely worth installing the HDF command-line tools. These are generally available through a package manager; if not, you can get them at www.hdfgroup.org. Windows versions are also available.

The program we'll be using most in this book is called `h5ls`, which as the name suggests lists the contents of a file. Here's an example, in which `h5ls` is applied to a file containing a couple of dataset and a group:

```
$ h5ls demo.hdf5
array                    Dataset {10}
group                    Group
scalar                   Dataset {SCALAR}
```

We can get a little more useful information by using the option combo `-vlr`, which prints extended information and also recursively enters groups:

```
$ h5ls -vlr demo.hdf5
/                        Group
    Location:  1:96
    Links:     1
/array                   Dataset {10/10}
    Location:  1:1400
    Links:     1
```

```
    Storage:   40 logical bytes, 40 allocated bytes, 100.00% utilization
    Type:      native int
/group                   Group
    Location:  1:1672
    Links:     1
/group/subarray          Dataset {2/2, 2/2}
    Location:  1:1832
    Links:     1
    Storage:   16 logical bytes, 16 allocated bytes, 100.00% utilization
    Type:      native int
/scalar                  Dataset {SCALAR}
    Location:  1:800
    Links:     1
    Storage:   4 logical bytes, 4 allocated bytes, 100.00% utilization
    Type:      native int
```

That's a little more useful. We can see that the object at `/array` is of type "native int," and is a 1D array 10 elements long. Likewise, there's a dataset inside the group named `group` that is 2D, also of type native int.

`h5ls` is great for inspecting metadata like this. There's also a program called `h5dump`, which prints data as well, although in a more verbose format:

```
$ h5dump demo.hdf5
HDF5 "demo.hdf5" {
GROUP "/" {
   DATASET "array" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 10 ) / ( 10 ) }
      DATA {
      (0): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
      }
   }
   GROUP "group" {
      DATASET "subarray" {
         DATATYPE  H5T_STD_I32LE
         DATASPACE  SIMPLE { ( 2, 2 ) / ( 2, 2 ) }
         DATA {
         (0,0): 2, 2,
         (1,0): 2, 2
         }
      }
   }
   DATASET "scalar" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SCALAR
      DATA {
      (0): 42
      }
   }
}
}
```

Your First HDF5 File

Before we get to groups and datasets, let's start by exploring some of the capabilities of the `File` object, which will be your entry point into the world of HDF5.

Here's the simplest possible program that uses HDF5:

```
>>> f = h5py.File("name.hdf5")
>>> f.close()
```

The `File` object is your starting point; it has methods that let you create new datasets or groups in the file, as well as more pedestrian properties such as `.filename` and `.mode`.

Speaking of `.mode`, HDF5 files support the same kind of read/write modes as regular Python files:

```
>>> f = h5py.File("name.hdf5", "w")   # New file overwriting any existing file
>>> f = h5py.File("name.hdf5", "r")   # Open read-only (must exist)
>>> f = h5py.File("name.hdf5", "r+")  # Open read-write (must exist)
>>> f = h5py.File("name.hdf5", "a")   # Open read-write (create if doesn't exist)
```

There's one additional HDF5-specific mode, which can save your bacon should you accidentally try to overwrite an existing file:

```
>>> f = h5py.File("name.hdf5", "w-")
```

This will create a new file, but fail if a file of the same name already exists. For example, if you're performing a long-running calculation and don't want to risk overwriting your output file should the script be run twice, you could open the file in `w-` mode:

```
>>> f = h5py.File("important_file.hdf5", "w-")
>>> f.close()
>>> f = h5py.File("important_file.hdf5", "w-")
IOError: unable to create file (File accessability: Unable to open file)
```

By the way, you're free to use Unicode filenames! Just supply a normal Unicode string and it will transparently work, assuming the underlying operating system supports the UTF-8 encoding:

```
>>> name = u"name_eta_\u03b7"
>>> f = h5py.File(name)
>>> print f.filename
name_eta_η
```

> **NOTE**
>
> You might wonder what happens if your program crashes with open files. If the program exits with a Python exception, don't worry! The HDF library will automatically close every open file for you when the application exits.