Francesc Alted • Scott Prater

PyTables User's Guide

A hierarchical database for Python

Release 0.8

Alted, Francesc:

PyTables User's Guide

A hierarchical database for Python
Release 0.8

Typeset by Francesc Alted and Scott Prater
Day of print: 2004, February, 2th

Contents

La sabiduría no vale la pena si no es posible servirse de ella para inventar una nueva manera de preparar los garbanzos. (Wisdom isn't worth anything if you can't use it to come up with a new way to cook garbanzos).

—A wise Catalan in "Cien años de soledad" Gabriel García Márquez

Chapter 1

Introduction

The goal of `PyTables` is to enable the end user to manipulate easily scientific data **tables** and **array** objects in a hierarchical structure. The foundation of the underlying hierarchical data organization is the excellent `HDF5` library (see NCSA).

It should be noted that this package is not intended to serve as a complete wrapper for the entire HDF5 API, but only to provide a flexible, *very Pythonic* tool to deal with (arbitrarily) large amounts of data (typically bigger than available memory) in tables and arrays organized in a hierarchical and persistent disk storage structure.

A table is defined as a collection of records whose values are stored in *fixed-length* fields. All records have the same structure and all values in each field have the same *data type*. The terms *fixed-length* and strict *data types* may seem to be a strange requirement for an interpreted language like Python, but they serve a useful function if the goal is to save very large quantities of data (such as is generated by many Internet services applications or scientific applications, for example) in an efficient manner that reduces demand on CPU time and I/O.

In order to emulate in Python records mapped to HDF5 C structs `PyTables` implements a special *metaclass* object so as to easily define all its fields and other properties. `PyTables` also provides a powerful interface to mine data in tables. Records in tables are also known in the `HDF5` naming scheme as *compound* data types.

For example, you can define arbitrary tables in Python simply by declaring a class with name field and types information, such as in the following example:

```
class Particle(IsDescription):
    name      = StringCol(16)   # 16-character String
    idnumber  = Int64Col()      # Signed 64-bit integer
    ADCcount  = UInt16Col()     # Unsigned short integer
    TDCcount  = UInt8Col()      # unsigned byte
    grid_i    = Int32Col()      # integer
    grid_j    = IntCol()        # integer (equivalent to Int32Col)
    pressure  = Float32Col(shape=(2,3)) # 2-D float array (single-precision)
    energy    = FloatCol(shape=(2,3,4)) # 3-D float array (double-precision)
```

You then pass this class to the table constructor, fill its rows with your values, and save (arbitrarily large) collections of them to a file for persistent storage. After that, the data can be retrieved and post-processed quite easily with `PyTables` or even with another `HDF5` application (in C, Fortran, Java or whatever language that provides a library to interface with HDF5).

Other important entities in `PyTables` are the *array* objects that are analogous to tables with the difference that all of their components are homogeneous. They come in different flavors, like *generic* (they provide a quick and fast way to deal with for numerical arrays), *enlargeable* (arrays can be extended in any single dimension) and *variable length* (each row in the array can have a different number of elements).

The next section describes the most interesting capabilities of `PyTables`.

1.1 Main Features

PyTables takes advantage of the powerful object orientation and introspection capabilities offered by Python to provide these features:

- *Support for table entities:* Allows the user to work with a large number of records, i.e. more than will fit into memory.
- *Appendable tables:* Supports adding records to already created tables. This can be done even between different Python sessions without copying the dataset or redefining its structure.
- *Multidimensional table cells:* You can declare a column to consist of general array cells as well as scalars, as the majority of relational databases allow.
- *Support for numerical arrays:* Numeric (see Ascher *et al.*) and numarray (see Greenfield *et al.*) arrays can be used as a useful complement of tables to store homogeneous table slices (such as selections of table columns).
- *Enlargeable arrays:* You can add new elements to existing arrays on disk in any dimension you want (but only one).
- *Variable length arrays:* The number of elements in these arrays can be variable from row to row. This provides a lot of flexibility when dealing with complex data.
- *Supports a hierarchical data model:* Allows the user to clearly structure all the data. PyTables builds up an *object tree* in memory that replicates the underlying file data structure. Access to the file objects is achieved by walking through and manipulating this object tree.
- *Support of files bigger than 2 GB:* PyTables automatically inherits this capability from the underlying HDF5 library (assuming your platform supports the C long long integer, or, on Windows, __int64).
- *Ability to read/modify generic HDF5 files:* PyTables can access a wide range of objects in generic HDF5 files, like compound type datasets (that can be mapped to Table objects), homogeneous datasets (that can be mapped to Array objects) or variable length record datasets (that can be mapped to VLArray objects). Besides, if a dataset is not supported, it will be mapped into a special UnImplemented class (see 4.9), that will let the user see that the data is there, although it would be unreachable (still, you will be able to access the attributes and some metadata in the dataset). With that, PyTables probably can access and *modify* most of the HDF5 files out there.
- *Data compression:* Supports data compression (using the **Zlib**, **LZO** and **UCL** compression libraries) out of the box. This is important when you have repetitive data patterns and don't want to spend time searching for an optimized way to store them (saving you time spent analyzing your data organization).
- *High performance I/O:* On modern systems storing large amounts of data, tables and array objects can be read and written at a speed only limited by the performance of the underlying I/O subsystem. Moreover, if your data is compressible, even that limit is surmountable!
- *Architecture-independent:* PyTables has been carefully coded (as has HDF5 itself) with little-endian/big-endian byte orderings issues in mind. In principle you can write a file on a big-endian machine (like a Sparc or MIPS) and read it on other little-endian machine (like an Intel or Alpha) without problems. In addition, it has been tested successfully with 64 bit platforms (Intel-64, MIPS, UltraSparc).

1.2 The Object Tree

The hierarchical model of the underlying HDF5 library allows PyTables to manage tables and arrays in a tree-like structure. In order to achieve this, an *object tree* entity is *dynamically* created imitating the HDF5 structure on disk. The HDF5 objects are read by walking through this object tree. You can get a good picture of what kind data is kept in the object by examining the *metadata* nodes.

The different nodes in the object tree are instances of `PyTables` classes. There are several types of classes, but the most important ones are the `Group` and the `Leaf` classes. `Group` instances (referred to as *groups* from now on) are a grouping structure containing instances of zero or more groups or leaves, together with supplementary metadata. `Leaf` instances (referred to as *leaves*) are containers for actual data and cannot contain further groups or leaves. The `Table`, `Array`, `EArray`, `VLArray` and `UnImplemented` classes are descendents of `Leaf`, and inherit all its properties.

Working with groups and leaves is similar in many ways to working with directories and files on a Unix filesystem. As is the case with Unix directories and files, objects in the object tree are often described by giving their full (or absolute) path names. In `PyTables` this full path can be specified either as string (such as `'/subgroup2/table3'`) or as a complete object path written in a format known as the *natural name* schema (such as `file.root.subgroup2.table3`).

Support for *natural naming* is a key aspect of `PyTables`. It means that the names of instance variables of the node objects are the same as the names of the element's children[1]. This is very *Pythonic* and intuitive in many cases. Check the tutorial section 3.1.6 for usage examples.

You should also be aware that not all the data present in a file is loaded into the object tree. Only the *metadata* (i.e. special data that describes the structure of the actual data) is loaded. The actual data is not read until you request it (by calling a method on a particular node). Using the object tree (the metadata) you can retrieve information about the objects on disk such as table names, titles, name columns, data types in columns, numbers of rows, or, in the case of arrays, the shapes, typecodes, etc. of the array. You can also search through the tree for specific kinds of data then read it and process it. In a certain sense, you can think of `PyTables` as a tool that applies the same introspection capabilities of Python objects to large amounts of data in persistent storage.

To better understand the dynamic nature of this object tree entity, let's start with a sample `PyTables` script (you can find it in `examples/objecttree.py`) to create a HDF5 file:

```
from tables import *

class Particle(IsDescription):
    identity = StringCol(length=22, dflt=" ", pos = 0)  # character String
    idnumber = Int16Col(1, pos = 1)  # short integer
    speed    = Float32Col(1, pos = 1)  # single-precision

# Open a file in "w"rite mode
fileh = openFile("objecttree.h5", mode = "w")
# Get the HDF5 root group
root = fileh.root

# Create the groups:
group1 = fileh.createGroup(root, "group1")
group2 = fileh.createGroup(root, "group2")

# Now, create an array in the root group
array1 = fileh.createArray(root, "array1", ["string", "array"], "String array")
# Create 2 new tables in group1
table1 = fileh.createTable(group1, "table1", Particle)
table2 = fileh.createTable("/group2", "table2", Particle)
# Create the last table in group2
array2 = fileh.createArray("/group1", "array2", [1,2,3,4])

# Now, fill the tables:
for table in (table1, table2):
    # Get the record object associated with the table:
    row = table.row
```

[1] I got this simple but powerful idea from the excellent `Objectify` module by David Mertz (see Mertz)

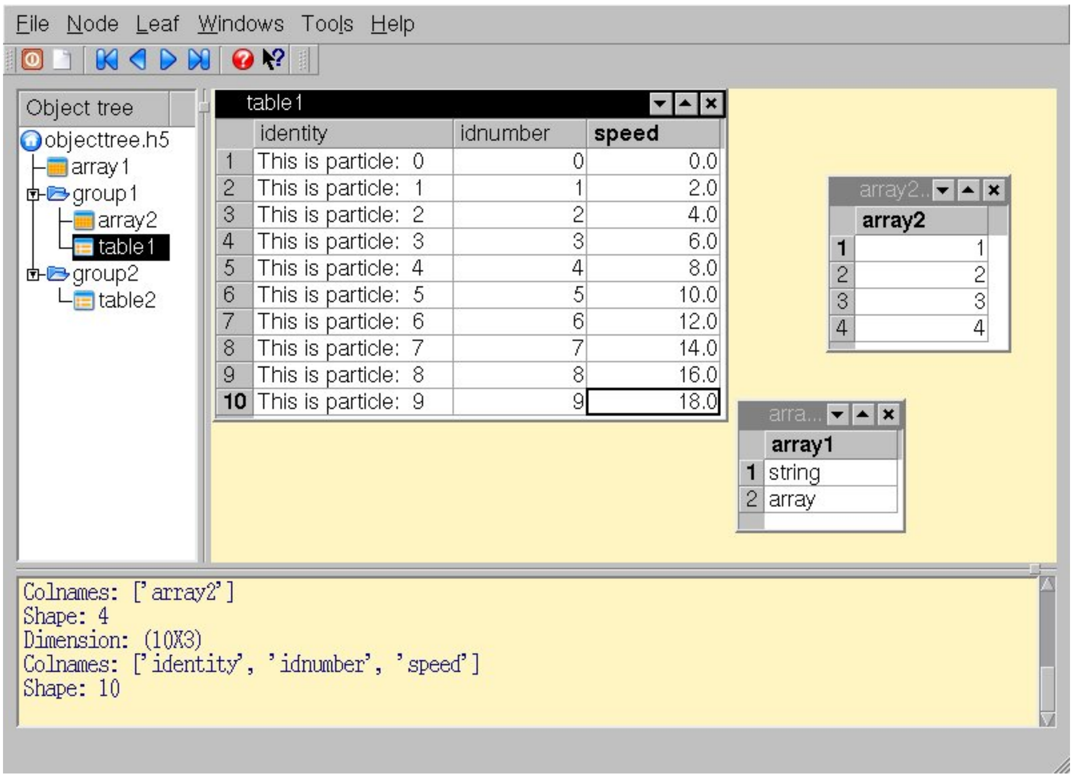

Figure 1.1: An HDF5 example with 2 subgroups, 2 tables and 1 array.

```
    # Fill the table with 10 records
    for i in xrange(10):
        # First, assign the values to the Particle record
        row['identity']  = 'This is particle: %2d' % (i)
        row['idnumber'] = i
        row['speed']   = i * 2.
        # This injects the Record values
        row.append()

    # Flush the table buffers
    table.flush()

# Finally, close the file (this also will flush all the remaining buffers!)
fileh.close()
```

This small program creates a simple HDF5 file called `objecttree.h5` with the structure that appears in figure 1.1. When the file is created, metadata in the object tree is updated in memory while the actual data is saved disk. When you close the file the object tree is no longer available. However, when you reopen this file the object tree will be reconstructed in memory from the metadata on disk, allowing you to work with it in exactly the same way as when you originally created it.

In figure 1.2 you can see an example of the object tree created when the above `objecttree.h5` file is read (in fact, such an object is always created when reading any supported generic HDF5 file). It's worthwhile to take your time to understand it[2]. It will help you to avoid programming mistakes.

[2] Bear in mind, however, that this diagram is **not** a standard UML class diagram; it is rather meant to show the connections between the `PyTables` objects and some of its most important attributes and methods.

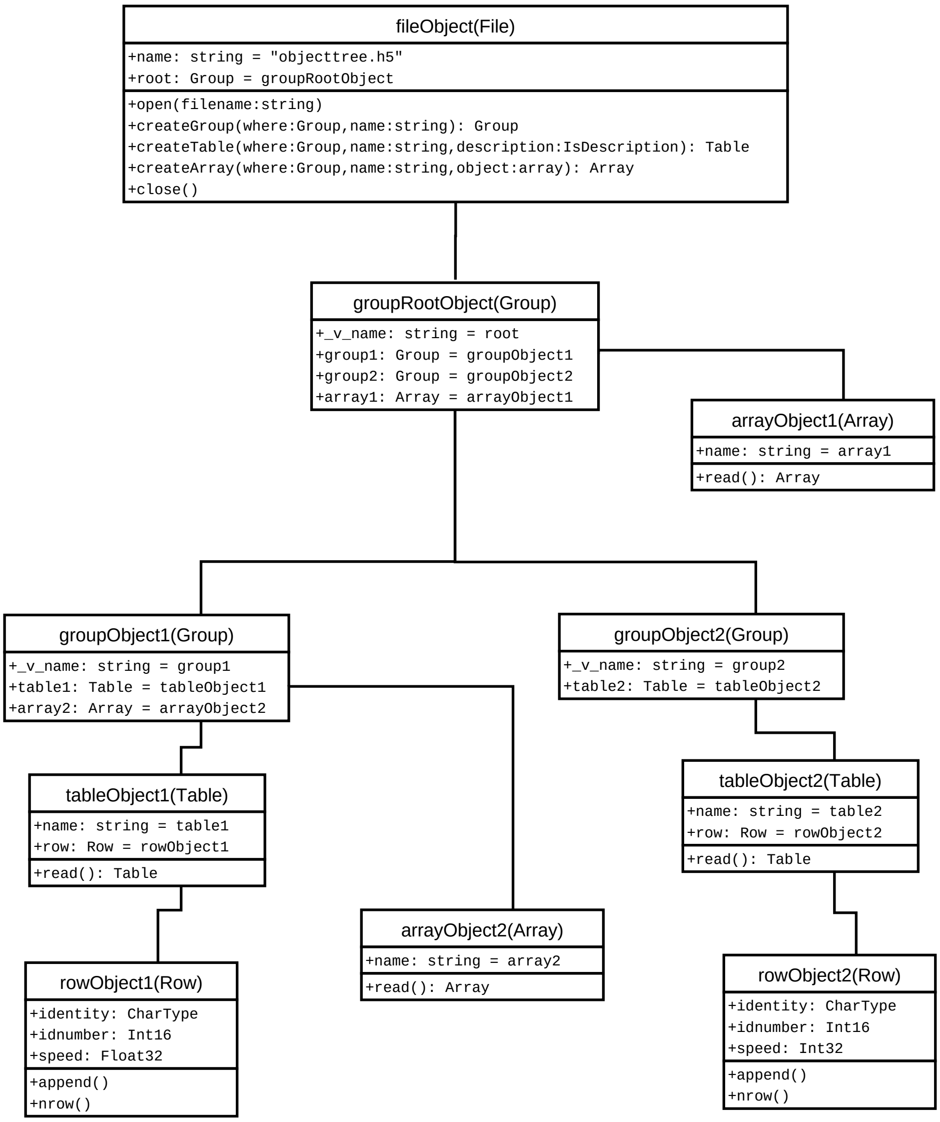

Figure 1.2: A `PyTables` object tree example.

Chapter 2

Installation

The Python Distutils are used to build and install PyTables, so it is fairly simple to get the application up and running. If you want to install the package from sources go to the next section. But if you are running Windows and want to install precompiled binaries jump to section 2.2). In addition, packages are starting to appear in different Linux distributions (like for instance RockLinux or Debian).

2.1 Installation from source

These instructions are for both Unix/Linux and Windows systems. If you are using Windows, it is assumed that you have a recent version of MS Visual C++ (>= 6.0) compiler installed. A GCC compiler is assumed for Unix, but other compilers should work as well.

Extensions in PyTables have developed in Pyrex (see Ewing) and C language. You can rebuild everything from scratch if you have Pyrex installed, but this is not necessary, as the Pyrex compiled source is included in the distribution.

To compile PyTables you will need a recent version of the HDF5 (C flavor) library and the numarray (see Greenfield *et al.*) package. Although you won't need Numerical Python (see Ascher *et al.*) in order to compile PyTables, it is supported; you only need a reasonably recent version of it (>= 21.x) if you plan on using its methods in your applications. PyTables has been successfully tested with Numeric 21.3, 22.0 and 23.0. If you already have Numeric installed, the test driver module will detect it and will run the tests for Numeric automatically.

2.1.1 Prerequisites

First, make sure that you have HDF5 1.6.2 and numarray 0.8 or higher installed (I'm using HDF5 1.6.2 and numarray 0.8 currently). If you don't, you can find them at http://hdf.ncsa.uiuc.edu/HDF5 and http://www.pfdubois.com/numpy.

Compile and install these packages (but see section 2.2.1 for instructions on how to install precompiled binaries if you are not willing to compile the prerequisites on Windows systems).

For compression (and possibly improved performance), you will need to install the Zlib (see Gailly and Adler), which is also required by HDF5 as well. You may also optionally install the excellent LZO and UCL compression libraries (see Oberhumer and section 5.2).

Unix setup.py will detect HDF5, LZO or UCL libraries and include files under /usr or /usr/local; this will cover most manual installations as well as installations from packages. If setup.py can't find libhdf5 or libz (or liblzo or libucl that you may wish to use) or if you have several versions of a library installed and want to use a particular one, then you can set the path to the resource in the environment, setting the values of the HDF5_DIR, LZO_DIR or UCL_DIR environment variables to the path to the particular resource. You may also specify the locations of the resource root directories on the setup.py command line. For example:

```
--hdf5=/stuff/hdf5-1.6.2
--lzo=/stuff/lzo-1.07
--ucl=/stuff/ucl-1.0.1
```

If your `HDF5` library was built as a shared library not in the runtime load path, then you can specify the additional linker flags needed to find the shared library on the command line as well. For example:

```
--lflags="-Xlinker -rpath -Xlinker /stuff/hdf5-1.6.2/lib"
```

or perhaps just

```
--rpath="/stuff/hdf5-1.6.2/lib"
```

Check your compiler and linker documentation as well as the Python `Distutils` documentation for the correct syntax.

It is also possible to link specific libraries with the standard `Distutils` `--libraries` switch:

```
--libraries="-lhdf5-1.6.5"
--libraries="-lhdf5-1.6.5 -lnsl"
```

Windows Once you have installed the prerequisites, `setup.py` needs to know where the necessary library *stub* (`.lib`) and *header* (`.h`) files are installed. Set the following environment variables:

HDF5_DIR Points to the root HDF5 directory (where the include/ and dll/ directories can be found). *Mandatory*.

LZO_DIR Points to the root LZO directory (where the include/ and lib/ directories can be found). *Optional*.

UCL_DIR Points to the root UCL directory (where the include/ and lib/ directories can be found). *Optional*.

For example:

```
set HDF5_DIR=c:\stuff\5-162-win2k\c\release
set UCL_DIR=c:\stuff\ucl-1-01
set LZO_DIR=c:\stuff\lzo-1-07
```

Or, you can pass this information to `setup.py` by setting the appropriate arguments on the command line. For example:

```
--hdf5=c:\stuff\5-162-win2k\c\release
--lzo=c:\stuff\lzo-1-07 --ucl=c:\stuff\ucl-1-01
```

2.1.2 `PyTables` package installation

Once you have installed the HDF5 library and numarray packages, you can proceed with the `PyTables` package itself:

1. Run this command from the main `PyTables` distribution directory, including any extra command line arguments as discussed above:

```
python setup.py build_ext --inplace
```

Depending on the compiler flags used when compiling your Python executable, there may appear many warnings. Don't worry, almost all of them are caused by variables declared but never used. That's normal in Pyrex extensions.

2. To run the test suite, change into the test directory and execute this command:

   Unix In the shell `sh` and its variants:

   ```
   PYTHONPATH=..
   export PYTHONPATH
   python test_all.py
   ```

   Windows Open a DOS terminal and type:

   ```
   set PYTHONPATH=..
   python test_all.py
   ```

   If you would like to see verbose output from the tests simply add the flag `-v` and/or the word `verbose` to the command line. You can also run only the tests in a particular test module. For example, to execute just the `types` test:

   ```
   python test_types.py -v
   ```

   If a test fails, please enable verbose output (the `-v` flag **and** `verbose` option), run the failing test module again, and, very important, get your `PyTables` version information by running the command:

   ```
   python test_all.py --show-versions-only
   ```

   and send back the output to developers so that we may continue improving `PyTables`.

   If you run into problems because Python can't load the HDF5 library or other shared libraries:

   Unix Try setting the LD_LIBRARY_PATH environment variable to point to the directory where the missing libraries can be found.

   Windows Put the DLL libraries (`hdf5dll.dll` and, optionally, `lzo.dll` and `ucl.dll`) in a directory listed in your `PATH` environment variable. The `setup.py` installation program will print out a warning to that effect if the libraries can't be found.

3. To install the entire `PyTables` Python package, change back to the root distribution directory and run the following command (make sure you have sufficient permissions to write to the directories where the `PyTables` files will be installed):

   ```
   python setup.py install
   ```

   Of course, you will need super-user privileges if you want to install `PyTables` on a system-protected area. You can select, though, a different place to install the package using the `--prefix` flag:

   ```
   python setup.py install --prefix="/home/myuser/mystuff"
   ```

   Have in mind, however, that if you use the `--prefix` flag to install in a non-standard place, you should properly setup your `PYTHONPATH` environment variable, so that the python interpreter would be able to find your new `PyTables` installation.

   You have more installation options available in distutils package. Issue a:

   ```
   python setup.py install --help
   ```

   for more information on that subject.

That's it! The next chapter describes how to use `PyTables`.

2.2 Binary installation (Windows)

This section is intended for installing precompiled binaries on Windows platforms. You may also find it useful for instructions on how to install *binary prerequisites* even if you want to compile `PyTables` itself on Windows.

2.2.1 Windows prerequisites

First, make sure that you have HDF5 1.6.2 or higher and numarray 0.8 or higher installed (I'm using HDF5 1.6.2 and numarray 0.8 currently). If don't, you can find them at `http://hdf.ncsa.uiuc.edu/HDF5` and `http://sourceforge.net/projects/numpy/`. Download the binary packages (or sources, if you want to compile everything yourself) and install them.

For the HDF5 it should be enough to manually copy the `hdf5dll.dll`, `zlib1.dll` **and** `szipdll.dll` files to a directory in your `PATH` environment variable (for example `C:\WINDOWS\SYSTEM`).

Caveat: When downloading the binary distribution for HDF5 libraries, select one compiled with MSVC 6.0, such as the package `5-162-win2k.zip`, regardless of whether you are using Win2k or WinXP (it should work fine on both). The file `5-162-winxp-net.zip` was compiled with the MSVC 7.0 (aka "`.NET`") and **does not** work well with the PyTables binary (which has been generated with MSVC 6.0). You have been warned!

To enable compression with optional LZO and UCL libraries (see the section 5.2 for hints about how they may be used to improve performance), fetch and install the `LZO` and `UCL` binaries from: `http://gnuwin32.sourceforge.net/`. Normally, you will only need to fetch and install the `<package>-<version>-bin.zip` file and copy the `lzo.dll` or `ucl.dll` files in a directory in the `PATH` environment variable, so that they can be found by the `PyTables` extensions.

Note: If you are reading this because you have been redirected from the section 2.1 (*Installation from source*), some of the headers you will need are in the `<package>-<version>-lib.zip` file.

2.2.2 `PyTables` package installation

Download the `tables-<version>.win32-py<version>.exe` (`tables-<version>-LU.win32-py<version>.exe` if you want support for LZO and UCL libraries) file and execute it.

You can (*you should*) test your installation by unpacking the source tar-ball, changing to the `test/` subdirectory and executing the `test_all.py` script. If all the tests pass (possibly with a few warnings, related to the potential unavailability of LZO and UCL libs) you already have a working, well-tested copy of `PyTables` installed! If any test fails, please try to locate which test module is failing and execute:

```
python test_<module>.py -v verbose
```

and also:

```
python test_all.py --show-versions-only
```

and mail the output to the developers so that the problem can be fixed in future releases.

That's it! Now, proceed to the next chapter to see how to use `PyTables`.

Chapter 3

Tutorials

This chapter consists of a series of simple yet comprehensive tutorials that will enable you to understand `PyTables`' main features. If you would like more information about some particular instance variable, global function, or method, look at the doc strings or go to the library reference in chapter 4. If you are reading this in PDF or HTML formats, follow the corresponding hyperlink near each newly introduced entity.

Please note that throughout this document the terms *column* and *field* will be used interchangeably, as will the terms *row* and *record*.

3.1 Getting started

In this section, we will see how to define our own records in Python and save collections of them (i.e. a **table**) into a file. Then we will select some of the data in the table using Python cuts and create `numarray` arrays to store this selection as separate objects in a tree.

In *examples/tutorial1-1.py* you will find the working version of all the code in this section. Nonetheless, this tutorial series has been written to allow you reproduce it in a Python interactive console. I encourage you to do parallel testing and inspect the created objects (variables, docs, children objects, etc.) during the course of the tutorial!

3.1.1 Importing `tables` objects

Before starting you need to import the public objects in the `tables` package. You normally do that by executing:

```
>>> import tables
```

This is the recommended way to import `tables` if you don't want to pollute your namespace. However, `PyTables` has a very reduced set of first-level primitives, so you may consider using the alternative:

```
>>> from tables import *
```

which will export in your caller application namespace the following objects: `openFile`, `isHDF5`, `isPyTablesFile` and `IsDescription`. This is a rather reduced set of objects, and for convenience, we will use this technique to access them.

If you are going to work with `numarray` or `Numeric` arrays (and normally, you will) you will also need to import objects from them. So most `PyTables` programs begin with:

```
>>> import tables        # but in this tutorial we use "from tables import *"
>>> from numarray import *  # or "from Numeric import *"
```

3.1.2 Declaring a Column Descriptor

Now, imagine that we have a particle detector and we want to create a table object in order to save data retrieved from it. You need first to define the table, the number of columns it has, what kind of object is contained in each column, and so on.

Our particle detector has a TDC (Time to Digital Converter) counter with a dynamic range of 8 bits and an ADC (Analogic to Digital Converter) with a range of 16 bits. For these values, we will define 2 fields in our record object called `TDCcount` and `ADCcount`. We also want to save the grid position in which the particle has been detected, so we will add two new fields called `grid_i` and `grid_j`. Our instrumentation also can obtain the pressure and energy of the particle. The resolution of the pressure-gauge allows us to use a simple-precision float to store `pressure` readings, while the `energy` value will need a double-precision float. Finally, to track the particle we want to assign it a name to identify the kind of the particle it is and a unique numeric identifier. So we will add two more fields: `name` will be a string of up to 16 characters, and `idnumber` will be an integer of 64 bits (to allow us to store records for extremely large numbers of particles).

Having determined our columns and their types, we can now declare a new `Particle` class that will contain all this information:

```
>>> class Particle(IsDescription):
...     name      = StringCol(16)   # 16-character String
...     idnumber  = Int64Col()      # Signed 64-bit integer
...     ADCcount  = UInt16Col()     # Unsigned short integer
...     TDCcount  = UInt8Col()      # unsigned byte
...     grid_i    = Int32Col()      # integer
...     grid_j    = IntCol()        # integer (equivalent to Int32Col)
...     pressure  = Float32Col()    # float  (single-precision)
...     energy    = FloatCol()      # double (double-precision)
...
>>>
```

This definition class is self-explanatory. Basically, you declare a class variable for each field you need. As its value you assign an instance of the appropriate `Col` subclass, according to the kind of column defined (the data type, the length, the shape, etc). See the section 4.11.2 for a complete description of these subclasses. See also appendix A for a list of data types supported by the `Col` constructor.

From now on, we can use `Particle` instances as a descriptor for our detector data table. We will see later on how to pass this object to construct the table. But first, we must create a file where all the actual data pushed into our table will be saved.

3.1.3 Creating a `PyTables` file from scratch

Use the first-level `openFile` (see 4.1.2) function to create a `PyTables` file:

```
>>> h5file = openFile("tutorial1.h5", mode = "w", title = "Test file")
```

`openFile` (see 4.1.2) is one of the objects imported by the "`from tables import *`" statement. Here, we are saying that we want to create a new file in the current working directory called "`tutorial1.h5`" in "`w`"rite mode and with an descriptive title string ("`Test file`"). This function attempts to open the file, and if successful, returns the `File` (see 4.2) object instance `h5file`. The root of the object tree is specified in the instance's `root` attribute.

3.1.4 Creating a new group

Now, to better organize our data, we will create a group called *detector* that branches from the root node. We will save our particle data table in this group.

```
>>> group = h5file.createGroup("/", 'detector', 'Detector information')
```

Here, we have taken the `File` instance `h5file` and invoked its `createGroup` method (see 4.2.2) to create a new group called *detector* branching from "/" (another way to refer to the `h5file.root` object we mentioned above). This will create a new `Group` (see 4.3) object instance that will be assigned to the variable `group`.

3.1.5 Creating a new table

Let's now create a `Table` (see 4.5) object as a branch off the newly-created group. We do that by calling the `createTable` (see 4.2.2) method of the `h5file` object:

```
>>> table = h5file.createTable(group, 'readout', Particle, "Readout example")
```

We create the `Table` instance under `group`. We assign this table the node name "*readout*". The `Particle` class declared before is the *description* parameter (to define the columns of the table) and finally we set "*Readout example*" as the `Table` title. With all this information, a new `Table` instance is created and assigned to the variable *table*.

If you are curious about how the object tree looks right now, simply `print` the `File` instance variable *h5file*, and examine the output:

```
>>> print h5file
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:00:13 2003'
/ (Group) 'Test file'
/detector (Group) 'Detector information'
/detector/readout (Table(0,)) 'Readout example'
```

As you can see, a dump of the object tree is displayed. It's easy to see the `Group` and `Table` objects we have just created. If you want more information, just type the variable containing the `File` instance:

```
>>> h5file
File(filename='tutorial1.h5', title='Test file', mode='w', trMap={}, rootUEP='/')
/ (Group) 'Test file'
/detector (Group) 'Detector information'
/detector/readout (Table(0,)) 'Readout example'
  description := {
    "ADCcount": Col('UInt16', shape=1, itemsize=2, dflt=0),
    "TDCcount": Col('UInt8', shape=1, itemsize= 1, dflt=0),
    "energy": Col('Float64', shape=1, itemsize=8, dflt=0.0),
    "grid_i": Col('Int32', shape=1, itemsize=4, dflt=0),
    "grid_j": Col('Int32', shape=1, itemsize=4, dflt=0),
    "idnumber": Col('Int64', shape=1, itemsize=8, dflt=0),
    "name": Col('CharType', shape=1, itemsize=16, dflt=None),
    "pressure": Col('Float32', shape=1, itemsize=4, dflt=0.0) }
  byteorder := little
```

More detailed information is displayed about each object in the tree. Note how `Particle`, our table descriptor class, is printed as part of the *readout* table description information. In general, you can obtain much more information about the objects and their children by just printing them. That introspection capability is very useful, and I recommend that you use it extensively.

The time has come to fill this table with some values. First we will get a pointer to the `Row` (see 4.5.4) instance of this `table` instance:

```
>>> particle = table.row
```

The `row` attribute of `table` points to the `Row` instance that will be used to write data rows into the table. We write data simply by assigning the `Row` instance the values for each row as if it were a dictionary (although it is actually an *extension class*), using the column names as keys.

Below is an example of how to write rows:

```
>>> for i in xrange(10):
...     particle['name']  = 'Particle: %6d' % (i)
...     particle['TDCcount'] = i % 256
...     particle['ADCcount'] = (i * 256) % (1 << 16)
...     particle['grid_i'] = i
...     particle['grid_j'] = 10 - i
...     particle['pressure'] = float(i*i)
...     particle['energy'] = float(particle['pressure'] ** 4)
...     particle['idnumber'] = i * (2 ** 34)
...     particle.append()
...
>>>
```

This code should be easy to understand. The lines inside the loop just assign values to the different columns in the Row instance `particle` (see 4.5.4). A call to its `append()` method writes this information to the `table` I/O buffer.

After we have processed all our data, we should flush the table's I/O buffer if we want to write all this data to disk. We achieve that by calling the `table.flush()` method.

```
>>> table.flush()
```

3.1.6 Reading (and selecting) data in a table

Ok. We have our data on disk, and now we need to access it and select from specific columns the values we are interested in. See the example below:

```
>>> table = h5file.root.detector.readout
>>> pressure = [ x['pressure'] for x in table.iterrows()
...              if x['TDCcount']>3 and 20<=x['pressure']<50 ]
>>> pressure
[25.0, 36.0, 49.0]
```

The first line creates a "shortcut" to the *readout* table deeper on the object tree. As you can see, we use the **natural naming** schema to access it. We also could have used the `h5file.getNode()` method, as we will do later on.

You will recognize the last two lines as a Python list comprehension. It loops over the rows in *table* as they are provided by the `table.iterrows()` iterator (see 4.5.2). The iterator returns values until all the data in table is exhausted. These rows are filtered using the expression `x['TDCcount'] > 3 and x['pressure'] < 50`. We select the value of the `pressure` column from filtered records to create the final list and assign it to `pressure` variable.

We could have used a normal `for` loop to accomplish the same purpose, but I find comprehension syntax to be more compact and elegant.

Let's select the `name` column for the same set of cuts:

```
>>> names=[ x['name'] for x in table if x['TDCcount']>3 and 20<=x['pressure']<50 ]
>>> names
['Particle:      5', 'Particle:      6', 'Particle:      7']
```

Note how we have omitted the `iterrows()` call in the list comprehension. The `Table` class has an implementation of the special method `__iter__()` that iterates over all the rows in the table. In fact, `iterrows()` internally calls this special `__iter__()` method. Accessing all the rows in a table using this method is very convenient, especially when working with the data interactively.

That's enough about selections. The next section will show you how to save these select results to a file.

3.1.7 Creating new array objects

In order to separate the selected data from the mass of detector data, we will create a new group `columns` branching off the root group. Afterwards, under this group, we will create two arrays that will contain the selected data. First, we create the group:

```
>>> gcolumns = h5file.createGroup(h5file.root, "columns", "Pressure and Name")
```

Note that this time we have specified the first parameter using *natural naming* (`h5file.root`) instead of with an absolute path string ("/").

Now, create the first of the two `Array` objects we've just mentioned:

```
>>> h5file.createArray(gcolumns, 'pressure', array(pressure),
...                      "Pressure column selection")
/columns/pressure (Array(3,)) 'Pressure column selection'
  type = Float64
  itemsize = 8
  flavor = 'NumArray'
  byteorder = 'little'
```

We already know the first two parameters of the `createArray` (see 4.2.2) methods (these are the same as the first two in `createTable`): they are the parent group *where* `Array` will be created and the `Array` instance *name*. The third parameter is the *object* we want to save to disk. In this case, it is a `Numeric` array that is built from the selection list we created before. The fourth parameter is the *title*.

Now, we will save the second array. It contains the list of strings we selected before: we save this object as-is, with no further conversion.

```
>>> h5file.createArray(gcolumns, 'name', names, "Name column selection")
/columns/name Array(4,) 'Name column selection'
  type = 'CharType'
  itemsize = 16
  flavor = 'List'
  byteorder = 'little'
```

As you can see, `createArray()` accepts *names* (which is a regular Python list) as an *object* parameter. Actually, it accepts a variety of different regular objects (see 4.2.2) as parameters. The `flavor` attribute (see the output above) saves the original kind of object that was saved. Based on this *flavor*, `PyTables` will be able to retrieve exactly the same object from disk later on.

Note that in these examples, the `createArray` method returns an `Array` instance that is not assigned to any variable. Don't worry, this is intentional to show the kind of object we have created by displaying its representation. The `Array` objects has been attached to the object tree and saved to disk, as you can see if you print the complete object tree:

```
>>> print h5file
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:00:13 2003'
/ (Group) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
```

```
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

3.1.8 Closing the file and looking at its content

To finish this first tutorial, we use the `close` method of the h5file `File` object to close the file before exiting Python:

```
>>> h5file.close()
>>> ^D
```

You have now created your first `PyTables` file with a table and two arrays. You can examine it with any generic HDF5 tool, such as `h5dump` or `h5ls`. Here is what the `tutorial1.h5` looks like when read with the `h5ls` program:

```
$ h5ls -rd tutorial1.h5
/columns                 Group
/columns/name            Dataset {3}
    Data:
        (0) "Particle:      5", "Particle:      6", "Particle:      7"
/columns/pressure        Dataset {3}
    Data:
        (0) 25, 36, 49
/detector                Group
/detector/readout        Dataset {10/Inf}
    Data:
        (0) {0, 0, 0, 0, 10, 0, "Particle:      0", 0},
        (1) {256, 1, 1, 1, 9, 17179869184, "Particle:      1", 1},
        (2) {512, 2, 256, 2, 8, 34359738368, "Particle:      2", 4},
        (3) {768, 3, 6561, 3, 7, 51539607552, "Particle:      3", 9},
        (4) {1024, 4, 65536, 4, 6, 68719476736, "Particle:      4", 16},
        (5) {1280, 5, 390625, 5, 5, 85899345920, "Particle:      5", 25},
        (6) {1536, 6, 1679616, 6, 4, 103079215104, "Particle:      6", 36},
        (7) {1792, 7, 5764801, 7, 3, 120259084288, "Particle:      7", 49},
        (8) {2048, 8, 16777216, 8, 2, 137438953472, "Particle:      8", 64},
        (9) {2304, 9, 43046721, 9, 1, 154618822656, "Particle:      9", 81}
```

Here's the outputs as displayed by the "ptdump" `PyTables` utility (located in `utils/` directory):

```
$ ptdump tutorial1.h5
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:40:51 2003'
/ (Group) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

You can pass the `-v` or `-d` options to `ptdump` if you want more verbosity. Try them out!

3.2 Browsing the *object tree* and appending to tables

In this section, we will learn how to browse the tree and retrieve meta-information about the actual data, then append some rows to an existing table to show how table objects can be enlarged.

In *examples/tutorial1-2.py* you will find the working version of all the code in this section. As before, you are encouraged to use a python shell and inspect the object tree during the course of the tutorial.

3.2.1 Traversing the object tree

Let's start by opening the file we created in last tutorial section.

```
>>> h5file = openFile("tutorial1.h5", "a")
```

This time, we have opened the file in "a"ppend mode. We use this mode to add more information to the file.

`PyTables`, following the Python tradition, offers powerful introspection capabilities, i.e. you can easily ask information about any component of the object tree as well as search the tree.

To start with, you can get a preliminary overview of the object tree by simply printing the existing `File` instance:

```
>>> print h5file
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:40:51 2003'
/ (Group) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

It looks like all of our objects are there. Now let's make use of the `File` iterator to see to list all the nodes in the object tree:

```
>>> for node in h5file:
...   print node
...
/ (Group) 'Test file'
/columns (Group) 'Pressure and Name'
/detector (Group) 'Detector information'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector/readout (Table(10,)) 'Readout example'
```

We can use the `walkGroups` method (see 4.2.2) of the `File` class to list only the *groups* on tree:

```
>>> for group in h5file.walkGroups("/"):
...   print group
...
/ (Group) 'Test file'
/columns (Group) 'Pressure and Name'
/detector (Group) 'Detector information'
```

Note that `walkGroups()` actually returns an *iterator*, not a list of objects. Using this iterator with the `listNodes()` method is a powerful combination. Let's see an example listing of all the arrays in the tree:

```
>>> for group in h5file.walkGroups("/"):
...     for array in h5file.listNodes(group, classname = 'Array'):
...         print array
...
/columns/name Array(4,) 'Name column selection'
/columns/pressure Array(4,) 'Pressure column selection'
```

`listNodes()` (see 4.2.2) returns a list containing all the nodes hanging off a specific `Group`. If the *classname* keyword is specified, the method will filter out all instances which are not descendants of the class. We have asked for only `Array` instances.

We can combine both calls by using the `__call__(where, classname)` special method of the `File` object (see 4.2.3). For example:

```
>>> for array in h5file("/", "Array"):
...   print array
...
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

This is a nice shortcut when working interactively.

Finally, we will list all the `Leaf`, i.e. `Table` and `Array`, instances (see 4.4 for detailed information on `Leaf` class), in the `/detector` group. Note that only one instance of the `Table` class (i.e. `readout`) will be selected in this group (as should be the case):

```
>>> for leaf in h5file.root.detector('Leaf'):
...   print leaf
...
/detector/readout (Table(10,)) 'Readout example'
```

We have used a call to the `Group.__call__(classname, recursive)` special method (4.3.3), using the *natural naming* path specification.

Of course you can do more sophisticated node selections using these powerful methods. But first, let's take a look at some important `PyTables` object instance variables.

3.2.2 Setting and getting user attributes

PyTables provides an easy and concise way to complement the meaning of your node objects on the tree by using the `AttributeSet` class (see section 4.10). You can access this object through the standard attribute `attrs` in `Leaf` nodes and `_v_attrs` in `Group` nodes.

For example, let's imagine that we want to save the date indicating when the data in `/detector/readout` table has been acquired, as well as the temperature during the gathering process:

```
>>> table = h5file.root.detector.readout
>>> table.attrs.gath_date = "Wed, 06/12/2003 18:33"
>>> table.attrs.temperature = 18.4
>>> table.attrs.temp_scale = "Celsius"
```

Now, let's set a somewhat more complex attribute in the `/detector` group:

```
>>> detector = h5file.root.detector
>>> detector._v_attrs.stuff = [5, (2.3, 4.5), "Integer and tuple"]
```

Note how the AttributeSet instance is accessed with the `_v_attrs` attribute because detector is a `Group` node. In general, you can save any standard Python data structure as an attribute node. See section 4.10 for a more detailed explanation of how they are serialized for export to disk.

Retrieving the attributes is equally simple:

```
>>> table.attrs.gath_date
'Wed, 06/12/2003 18:33'
>>> table.attrs.temperature
18.399999999999999
>>> table.attrs.temp_scale
'Celsius'
>>> detector._v_attrs.stuff
[5, (2.2999999999999998, 4.5), 'Integer and tuple']
```

You can probably guess how to delete attributes:

```
>>> del table.attrs.gath_date
```

If you want to examine the current complete attribute set of `/detector/table`, you can print its representation (try hitting the `TAB` key twice if you are on a Unix Python console with the `rlcompleter` module active):

```
>>> table.attrs
/detector/readout (AttributeSet), 14 attributes:
   [CLASS := 'TABLE',
    FIELD_0_NAME := 'ADCcount',
    FIELD_1_NAME := 'TDCcount',
    FIELD_2_NAME := 'energy',
    FIELD_3_NAME := 'grid_i',
    FIELD_4_NAME := 'grid_j',
    FIELD_5_NAME := 'idnumber',
    FIELD_6_NAME := 'name',
    FIELD_7_NAME := 'pressure',
    NROWS := 10,
    TITLE := 'Readout example',
    VERSION := '2.0',
    tempScale := 'Celsius',
    temperature := 18.399999999999999]
```

You can get a list of only the user or system attributes with the `_f_list()` method.

```
>>> print table.attrs._f_list("user")
['temp_scale', 'temperature']
>>> print table.attrs._f_list("sys")
['CLASS', 'FIELD_0_NAME', 'FIELD_1_NAME', 'FIELD_2_NAME', 'FIELD_3_NAME',
 'FIELD_4_NAME', 'FIELD_5_NAME', 'FIELD_6_NAME', 'FIELD_7_NAME', 'NROWS',
 'TITLE', 'VERSION']
```

You can also rename attributes:

```
>>> table.attrs._f_rename("temp_scale","tempScale")
>>> print table.attrs._f_list()
['tempScale', 'temperature']
```

However, you can’t set, delete or rename read-only attributes:

```
>>> table.attrs._f_rename("VERSION", "version")
Traceback (most recent call last):
  File ">stdin>", line 1, in ?
  File "/home/falted/PyTables/pytables-0.7/tables/AttributeSet.py", line 249, in _f_rename
    raise RuntimeError, \
RuntimeError: Read-only attribute ('VERSION') cannot be renamed
```

After your terminating your session, you can use `h5ls` to read the `/detector/readout` attributes from the file written to disk:

```
$ h5ls -vr tutorial1.h5/detector/readout
Opened "tutorial1.h5" with sec2 driver.
/detector/readout        Dataset {10/Inf}
    Attribute: CLASS     scalar
        Type:      6-byte null-terminated ASCII string
        Data:  "TABLE"
    Attribute: VERSION   scalar
        Type:      4-byte null-terminated ASCII string
        Data:  "2.0"
    Attribute: TITLE     scalar
        Type:      16-byte null-terminated ASCII string
        Data:  "Readout example"
    Attribute: FIELD_0_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "ADCcount"
    Attribute: FIELD_1_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "TDCcount"
    Attribute: FIELD_2_NAME scalar
        Type:      7-byte null-terminated ASCII string
        Data:  "energy"
    Attribute: FIELD_3_NAME scalar
        Type:      7-byte null-terminated ASCII string
        Data:  "grid_i"
    Attribute: FIELD_4_NAME scalar
        Type:      7-byte null-terminated ASCII string
        Data:  "grid_j"
    Attribute: FIELD_5_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "idnumber"
    Attribute: FIELD_6_NAME scalar
        Type:      5-byte null-terminated ASCII string
        Data:  "name"
    Attribute: FIELD_7_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "pressure"
    Attribute: tempScale scalar
        Type:      8-byte null-terminated ASCII string
        Data:  "Celsius"
    Attribute: temperature {1}
        Type:      native double
        Data:  18.4
    Attribute: NROWS     {1}
```

```
        Type:      native int
        Data:  10
    Location:  0:1:0:1952
    Links:     1
    Modified:  2003-07-24 13:59:19 CEST
    Chunks:    {2048} 96256 bytes
    Storage:   470 logical bytes, 96256 allocated bytes, 0.49% utilization
    Type:      struct {
                   "ADCcount"         +0    native unsigned short
                   "TDCcount"         +2    native unsigned char
                   "energy"           +3    native double
                   "grid_i"           +11   native int
                   "grid_j"           +15   native int
                   "idnumber"         +19   native long long
                   "name"             +27   16-byte null-terminated ASCII string
                   "pressure"         +43   native float
               } 47 bytes
```

Attributes are a useful mechanism to add persistent (meta) information to your data.

3.2.3 Getting object metadata

Each object in `PyTables` has *metadata* information about the data in the file. Normally this *metainformation* is accessible through the node instance variables. Let's take a look at some examples:

```
>>> print "Object:", table
Object: /detector/readout Table(10,) 'Readout example'
>>> print "Table name:", table.name
Table name: readout
>>> print "Table title:", table.title
Table title: Readout example
>>> print "Number of rows in table:", table.nrows
Number of rows in table: 10
>>> print "Table variable names with their type and shape:"
Table variable names with their type and shape:
>>> for name in table.colnames:
...   print name, ':= %s, %s' % (table.coltypes[name], table.colshapes[name])
...
ADCcount := UInt16, 1
TDCcount := UInt8, 1
energy := Float64, 1
grid_i := Int32, 1
grid_j := Int32, 1
idnumber := Int64, 1
name := CharType, 1
pressure := Float32, 1
```

Here, the `name`, `title`, `nrows`, `colnames`, `coltypes` and `colshapes` attributes (see 4.2.1 for a complete attribute list) of the `Table` object gives us quite a bit of information about the table data.

You can interactively retrieve general information about the public objects in PyTables by printing their internal doc strings:

```
>>> print table.__doc__
Represent a table in the object tree.

    It provides methods to create new tables or open existing ones, as
    well as to write/read data to/from table objects over the
    file. A method is also provided to iterate over the rows without
    loading the entire table or column in memory.

    Data can be written or read both as Row instances or as numarray
    (NumArray or RecArray) objects.

    Methods:

      Common to all leaves:
        close()
        flush()
        getAttr(attrname)
        rename(newname)
        remove()
        setAttr(attrname, attrvalue)

      Specific of Table:
        iterrows()
        read([start] [, stop] [, step] [, field [, flavor]])
        removeRows(start, stop)

    Instance variables:

      Common to all leaves:
        name -- the leaf node name
        hdf5name -- the HDF5 leaf node name
        title -- the leaf title
        shape -- the leaf shape
        byteorder -- the byteorder of the leaf

      Specific of Table:
        description -- the metaobject describing this table
        row -- a reference to the Row object associated with this table
        nrows -- the number of rows in this table
        rowsize -- the size, in bytes, of each row
        colnames -- the field names for the table (list)
        coltypes -- the type class for the table fields (dictionary)
        colshapes -- the shapes for the table fields (dictionary)
```

The `help` function is also a handy way to see `PyTables` reference documentation online. Try it yourself with other object docs:

```
>>> help(table.__class__)
>>> help(table.removeRows)
```

To examine metadata in the */columns/pressure* `Array` object:

```
>>> pressureObject = h5file.getNode("/columns", "pressure")
```

```
>>> print "Info on the object:", repr(pressureObject)
Info on the object: /columns/pressure (Array(3,)) 'Pressure column selection'
  type = Float64
  itemsize = 8
  flavor = 'NumArray'
  byteorder = 'little'
>>> print "  shape: ==>", pressureObject.shape
  shape: ==> (3,)
>>> print "  title: ==>", pressureObject.title
  title: ==> Pressure column selection
>>> print "  type: ==>", pressureObject.type
  type: ==> Float64
```

Observe that we have used the `getNode()` method of the `File` class to access a node in the tree, instead of the natural naming method. Both are useful, and depending on the context you will prefer one or the other. `getNode()` has the advantages that it can get a node from the pathname string (as in this example) and can also act as a filter to show only nodes in a particular location that are instances of class *classname*. In general, however, I consider natural naming to be more elegant and easier to use, especially if you are using the name completion capability present in interactive console. Try this powerful combination of natural naming and completion capabilities present in most Python consoles, and see how pleasant it is to browse the object tree (at least, as pleasant as such an activity can be).

If you look at the `type` attribute of the `pressureObject` object, you can verify that it is a "**Float64**" array. By looking at its `shape` attribute, you can deduce that the array on disk is unidimensional and has 4 elements. See 4.6.1 or the internal string docs for the complete `Array` attribute list.

3.2.4 Reading data from `Array` objects

Once you have found the desired `Array`, use the `read()` method of the `Array` object to retrieve its data:

```
>>> pressureArray = pressureObject.read()
>>> pressureArray
array([ 25.,  36.,  49.])
>>> print "pressureArray is an object of type:", type(pressureArray)
pressureArray is an object of type: <class 'numarray.numarraycore.NumArray'>
>>> nameArray = h5file.root.columns.name.read()
>>> nameArray
['Particle:      5', 'Particle:      6', 'Particle:      7']
>>> print "nameArray is an object of type:", type(nameArray)
nameArray is an object of type: <type 'list'>
>>>
>>> print "Data on arrays nameArray and pressureArray:"
Data on arrays nameArray and pressureArray:
>>> for i in range(pressureObject.shape[0]):
...   print nameArray[i], "-->", pressureArray[i]
...
Particle:      5 --> 25.0
Particle:      6 --> 36.0
Particle:      7 --> 49.0
>>> pressureObject.name
'pressure'
```

You can see that the `read()` method (see section 4.6.2) returns an authentic `numarray` object for the `pressureObject` instance by looking at the output of the `type()` call. A `read()` of the `nameObject`

object instance returns a native Python list (of strings). The type of the object saved is stored as an HDF5 attribute (named `FLAVOR`) for objects on disk. This attribute is then read as `Array` metainformation (accessible through in the `Array.attrs.FLAVOR` variable), enabling the read array to be converted into the original object. This provides a means to save a large variety of objects as arrays with the guarantee that you will be able to later recover them in their original form. See section 4.2.2 for a complete list of supported objects for the `Array` object class.

3.2.5 Appending data to an existing table

Now, let's have a look at how we can add records to an existing table on disk. Let's use our well-known *readout* `Table` object and append some new values to it:

```
>>> table = h5file.root.detector.readout
>>> particle = table.row
>>> for i in xrange(10, 15):
...     particle['name']  = 'Particle: %6d' % (i)
...     particle['TDCcount'] = i % 256
...     particle['ADCcount'] = (i * 256) % (1 << 16)
...     particle['grid_i'] = i
...     particle['grid_j'] = 10 - i
...     particle['pressure'] = float(i*i)
...     particle['energy'] = float(particle['pressure'] ** 4)
...     particle['idnumber'] = i * (2 ** 34)
...     particle.append()
...
>>> table.flush()
```

It's the same method we used to fill a new table. `PyTables` knows that this table is on disk, and when you add new records, they are appended to the end of the table[1].

If you look carefully at the code you will see that we have used the `table.row` attribute to create a table row and fill it with the new values. Each time that its `append()` method is called, the actual row is committed to the output buffer and the row pointer is incremented to point to the next table record. When the buffer is full, the data is saved on disk, and the buffer is reused again for the next cycle.

Caveat emptor: Do not forget to always call the `.flush()` method after a write operation, or else your tables will not be updated!

Let's have a look at some rows in the modified table and verify that our new data has been appended:

```
>>> for r in table.iterrows():
...     print "%-16s | %11.1f | %11.4g | %6d | %6d | %8d |" % \
...        (r['name'], r['pressure'], r['energy'], r['grid_i'], r['grid_j'],
...         r['TDCcount'])
...
...
Particle:      0 |         0.0 |           0 |      0 |     10 |        0 |
Particle:      1 |         1.0 |           1 |      1 |      9 |        1 |
Particle:      2 |         4.0 |         256 |      2 |      8 |        2 |
Particle:      3 |         9.0 |        6561 |      3 |      7 |        3 |
Particle:      4 |        16.0 |   6.554e+04 |      4 |      6 |        4 |
Particle:      5 |        25.0 |   3.906e+05 |      5 |      5 |        5 |
Particle:      6 |        36.0 |    1.68e+06 |      6 |      4 |        6 |
Particle:      7 |        49.0 |   5.765e+06 |      7 |      3 |        7 |
Particle:      8 |        64.0 |   1.678e+07 |      8 |      2 |        8 |
Particle:      9 |        81.0 |   4.305e+07 |      9 |      1 |        9 |
```

[1] Note that you can append not only scalar values to tables, but also fully multidimensional array objects.

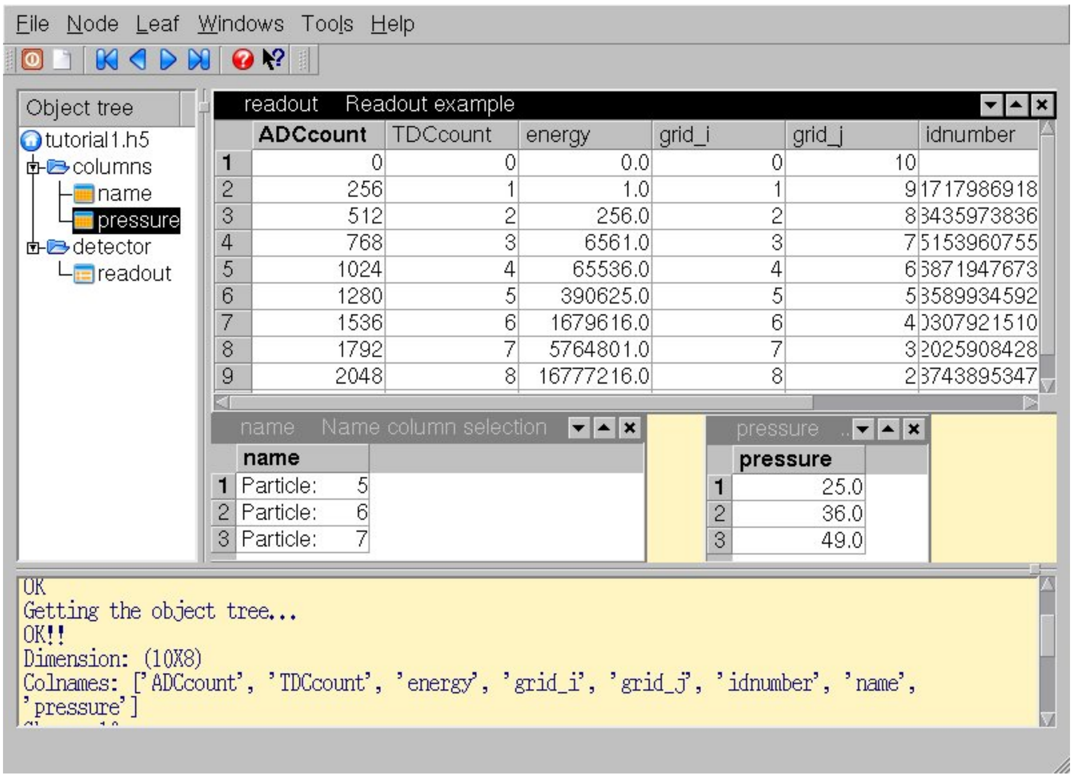

Figure 3.1: The final version of the data file for tutorial 1, with a view of the data objects.

```
Particle:     10 |      100.0 |      1e+08 |     10 |      0 |      10 |
Particle:     11 |      121.0 |  2.144e+08 |     11 |     -1 |      11 |
Particle:     12 |      144.0 |    4.3e+08 |     12 |     -2 |      12 |
Particle:     13 |      169.0 |  8.157e+08 |     13 |     -3 |      13 |
Particle:     14 |      196.0 |  1.476e+09 |     14 |     -4 |      14 |
```

3.2.6 And finally... how to delete rows from a table

We'll finish this tutorial by deleting some rows from the table we have. Suppose that we want to delete the the 5th to 9th rows (inclusive):

```
>>> table.removeRows(5,10)
5
```

`removeRows(start, stop)` (see 4.5.2) deletes the rows in the range (start, stop). It returns the number of rows effectively removed.

We have reached the end of this first tutorial. Don't forget to close the file when you finish:

```
>>> h5file.close()
>>> ^D
$
```

In figure 3.1 you can see a graphical view of the `PyTables` file with the datasets we have just created. In figure 3.2 are displayed the general properties of the table `/detector/readout`.

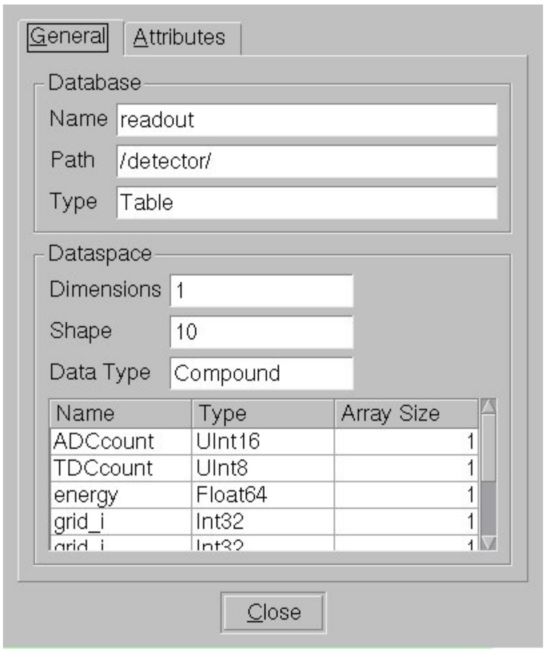

Figure 3.2: General properties of the `/detector/readout` table.

3.3 Multidimensional table cells and automatic sanity checks

Now it's time for a more real-life example (i.e. with errors in the code). We will create two groups that branch directly from the `root` node, `Particles` and `Events`. Then, we will put three tables in each group. In `Particles` we will put tables based on the `Particle` descriptor and in `Events`, the tables based the `Event` descriptor.

Afterwards, we will provision the tables with a number of records. Finally, we will read the newly-created table `/Events/TEvent3` and select some values from it, using a comprehension list.

Look at the next script (you can find it in `examples/tutorial2.py`). It appears to do all of the above, but it contains some small bugs. Note that this `Particle` class is not directly related to the one defined in last tutorial; this class is simpler (note, however, the *multidimensional* columns called `pressure` and `temperature`).

We also introduce a new manner to describe a `Table` as a dictionary, as you can see in the `Event` description. See section 4.2.2 about the different kinds of descriptor objects that can be passed to the `createTable()` method.

```
from numarray import *
from tables import *

# Describe a particle record
class Particle(IsDescription):
    name        = StringCol(length=16) # 16-character String
    lati        = IntCol()             # integer
    longi       = IntCol()             # integer
    pressure    = Float32Col(shape=(2,3)) # array of floats (single-precision)
    temperature = FloatCol(shape=(2,3))   # array of doubles (double-precision)

# Another way to describe the columns of a table
Event = {
    "name"    : Col('CharType', 16),    # 16-character String
    "TDCcount": Col("UInt8", 1),        # unsigned byte
```

```
    "ADCcount": Col("UInt16", 1),       # Unsigned short integer
    "xcoord"  : Col("Float32", 1),      # integer
    "ycoord"  : Col("Float32", 1),      # integer
    }

# Open a file in "w"rite mode
fileh = openFile("tutorial2.h5", mode = "w")
# Get the HDF5 root group
root = fileh.root
# Create the groups:
for groupname in ("Particles", "Events"):
    group = fileh.createGroup(root, groupname)
# Now, create and fill the tables in the Particles group
gparticles = root.Particles
# Create 3 new tables
for tablename in ("TParticle1", "TParticle2", "TParticle3"):
    # Create a table
    table = fileh.createTable("/Particles", tablename, Particle,
                           "Particles: "+tablename)
    # Get the record object associated with the table:
    particle = table.row
    # Fill the table with data for 257 particles
    for i in xrange(257):
        # First, assign the values to the Particle record
        particle['name'] = 'Particle: %6d' % (i)
        particle['lati'] = i
        particle['longi'] = 10 - i
        ########### Detectable errors start here. Play with them!
        particle['pressure'] = array(i*arange(2*3), shape=(2,4))  # Incorrect
        #particle['pressure'] = array(i*arange(2*3), shape=(2,3))  # Correct
        ########### End of errors
        particle['temperature'] = (i**2)     # Broadcasting
        # This injects the Record values
        particle.append()
    # Flush the table buffers
    table.flush()

# Now Events:
for tablename in ("TEvent1", "TEvent2", "TEvent3"):
    # Create a table in the Events group
    table = fileh.createTable(root.Events, tablename, Event,
                           "Events: "+tablename)
    # Get the record object associated with the table:
    event = table.row
    # Fill the table with data on 257 events
    for i in xrange(257):
        # First, assign the values to the Event record
        event['name']  = 'Event: %6d' % (i)
        event['TDCcount'] = i % (1<<8)   # Correct range
        ########### Detectable errors start here. Play with them!
        #event['xcoord'] = float(i**2)   # Correct spelling
        event['xcoor'] = float(i**2)     # Wrong spelling
        event['ADCcount'] = i * 2        # Correct type
        #event['ADCcount'] = "s"          # Wrong type
```

```
        ########### End of errors
        event['ycoord'] = float(i)**4
        # This injects the Record values
        event.append()

    # Flush the buffers
    table.flush()

# Read the records from table "/Events/TEvent3" and select some
table = root.Events.TEvent3
e = [ p['TDCcount'] for p in table
      if p['ADCcount'] < 20 and 4 <= p['TDCcount'] < 15 ]
print "Last record ==>", p
print "Selected values ==>", e
print "Total selected records ==> ", len(e)
# Finally, close the file (this also will flush all the remaining buffers)
fileh.close()
```

3.3.1 Shape checking

If you look at the code carefully, you'll see that it won't work. You will get the following error:

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 53, in ?
    particle['pressure'] = array(i*arange(2*3), shape=(2,4))  # Incorrect
  File "/usr/local/lib/python2.2/site-packages/numarray/numarraycore.py", line 281, in arr
    a.setshape(shape)
  File "/usr/local/lib/python2.2/site-packages/numarray/generic.py", line 530, in setshape
    raise ValueError("New shape is not consistent with the old shape")
ValueError: New shape is not consistent with the old shape
```

This error indicates that you are trying to assign an array with an incompatible shape to a table cell. Looking at the source, we see that we were trying to assign an array of shape `(2,4)` to a `pressure` element, which was defined with the shape `(2,3)`.

In general, these kinds of operations are forbidden, with one valid exception: when you assign a *scalar* value to a multidimensional column cell, all the cell elements are populated with the value of the scalar. For example:

```
        particle['temperature'] = (i**2)    # Broadcasting
```

The value `i**2` is assigned to all the elements of the `temperature` table cell. This capability is provided by the `numarray` package and is known as *broadcasting*.

3.3.2 Field name checking

After fixing the previous error and rerunning the program, we encounter another error:

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 74, in ?
    event['xcoor'] = float(i**2)     # Wrong spelling
  File "/home/falted/PyTables/pytables-0.7/src/hdf5Extension.pyx",
 line 1812, in hdf5Extension.Row.__setitem__
    raise AttributeError, "Error setting \"%s\" attr.\n %s" % \
AttributeError: Error setting "xcoor" attr.
```

```
 Error was: "exceptions.KeyError: xcoor"
```

This error indicates that we are attempting to assign a value to a non-existent field in the *event* table object. By looking carefully at the `Event` class attributes, we see that we misspelled the `xcoord` field (we wrote `xcoor` instead). This is unusual behavior for Python, as normally when you assign a value to a non-existent instance variable, Python creates a new variable with that name. Such a feature can be dangerous when dealing with an object that contains a fixed list of field names. PyTables checks that the field exists and raises a `KeyError` if the check fails.

3.3.3 Data type checking

Finally, in order to test type checking, we will change the next line:

```
    event.ADCcount = i * 2        # Correct type
```

to read:

```
    event.ADCcount = "s"          # Wrong type
```

This modification will cause the following `TypeError` exception to be raised when the script is executed:

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 76, in ?
    event['ADCcount'] = "s"          # Wrong type
  File "/home/falted/PyTables/pytables-0.7/src/hdf5Extension.pyx",
 line 1812, in hdf5Extension.Row.__setitem__
    raise AttributeError, "Error setting \"%s\" attr.\n %s" % \
AttributeError: Error setting "ADCcount" attr.
 Error was: "exceptions.TypeError: NA_setFromPythonScalar: bad value type."
```

You can see the structure created with this (corrected) script in figure 3.3. In particular, note the multidimensional column cells in table `/Particles/TParticle2`.

Feel free to examine the rest of examples in directory `examples`, and try to understand them. I've written several practical sample scripts to give you an idea of the `PyTables` capabilities, its way of dealing with HDF5 objects, and how it can be used in the real world.

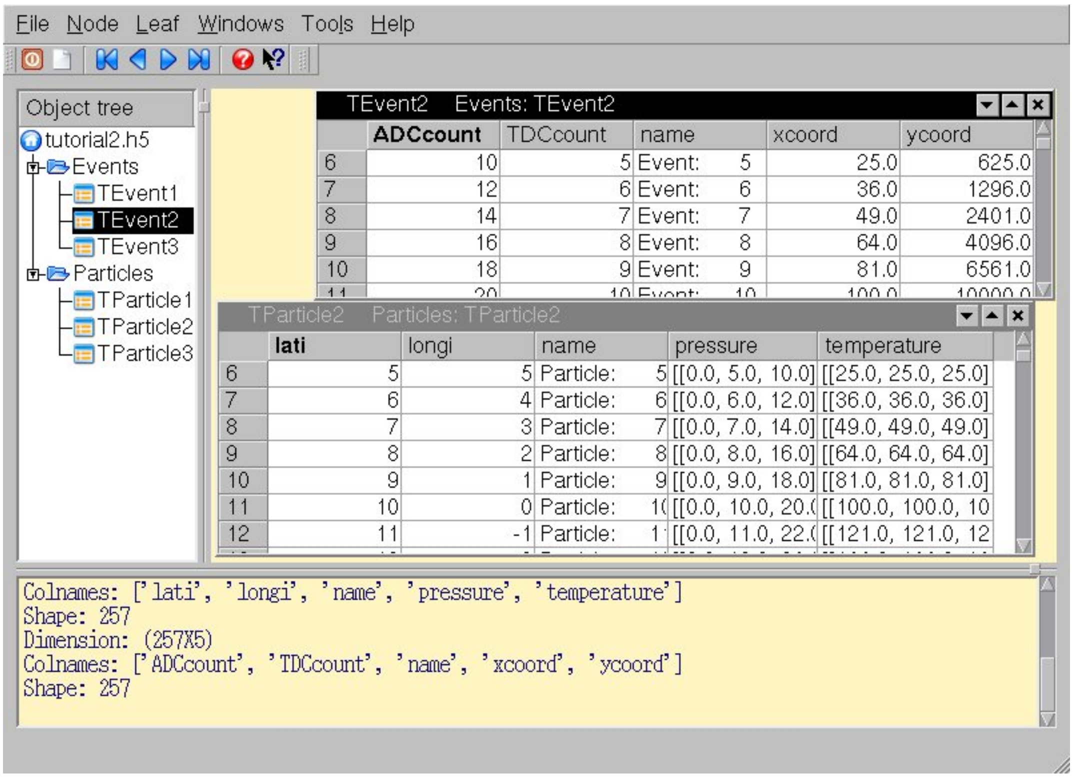

Figure 3.3: Table hierarchy for tutorial 2.

Chapter 4

Library Reference

`PyTables` implements several classes to represent the different nodes in the object tree. They are named `File`, `Group`, `Leaf`, `Table`, `Array`, `EArray`, `VLArray` and `UnImplemented`. Another one allows the user to complement the information on these different objects; its name is `AttributeSet`. Finally, another important class called `IsDescription` allows to build a `Table` record description by declaring a subclass of it. Many other classes are defined in `PyTables`, but they can be regarded as helpers whose goal is mainly to declare the *data type properties* of the different first class objects and will be described at the end of this chapter as well.

An important function, called `openFile` is responsible to create, open or append to files. In addition, a few utility functions are defined to guess if the user supplied file is a *PyTables* or *HDF5* file. These are called `isPyTablesFile` and `isHDF5`, respectively. Finally, there exists a function called `whichLibVersion` that informs about the versions of the underlying C libraries (for example, the `HDF5` or the `Zlib`).

Let's start discussing the first-level variables and functions available to the user, then the different classes defined in `PyTables`.

4.1 `tables` variables and functions

4.1.1 Global variables

__version__ The `PyTables` version number.

ExtVersion The version of the Pyrex extension module. This might be useful when reporting bugs.

HDF5Version The underlying HDF5 library version number.

4.1.2 Global functions

isHDF5(filename) Determines whether filename is in the HDF5 format or not. When successful, returns a positive value, for TRUE, or 0 (zero), for FALSE. Otherwise returns a negative value. To this function to work, it needs a closed file.

isPyTablesFile(filename) Determines whether a file is in the `PyTables` format. When successful, returns the format version string, for TRUE, or 0 (zero), for FALSE. Otherwise returns a negative value. To this function to work, it needs a closed file.

openFile(filename, mode='r', title='', trMap={}, rootUEP="/", filters=None) Open a `PyTables` (or generic `HDF5`) file and returns a `File` object.

- **filename** The name of the file (supports environment variable expansion). It is suggested that it should have any of `".h5"`, `".hdf"` or `".hdf5"` extensions, although this is not mandatory.
- **mode** The mode to open the file. It can be one of the following:

- **'r'** read-only; no data can be modified.
- **'w'** write; a new file is created (an existing file with the same name would be deleted).
- **'a'** append; an existing file is opened for reading and writing, and if the file does not exist it is created.
- **'r+'** is similar to 'a', but the file must already exist.

title If filename is new, this will set a title for the root group in this file. If filename is not new, the title will be read from disk, and this will not have any effect.

trMap A dictionary to map names in the object tree Python namespace into different HDF5 names in file namespace. The keys are the Python names, while the values are the HDF5 names. This is useful when you need to use HDF5 node names with invalid or reserved words in Python.

rootUEP The root User Entry Point. This is a group in the HDF5 hierarchy which will be taken as the starting point to create the object tree. The group has to be named after its HDF5 name and can be a path. If it does not exist, a `RuntimeError` exception is issued. Use this if you do not want to build the **entire** object tree, but rather only a **subtree** of it.

filters An instance of the `Filters` class (see section 4.12.1) that provides information about the desired I/O filters applicable to the leaves that hangs directly from *root* (unless other filters properties are specified for these leaves). Besides, if you do not specify filter properties for its child groups, they will inherit these ones. So, if you open a new file with this parameter set, all the leaves that would be created in the file will recursively inherit this filtering properties (again, if you don't prevent that from happening by specifying other filters on the child groups or leaves).

whichLibVersion(libname) Returns info about versions of the underlying C libraries. **libname** can be whether `"hdf5"`, `"zlib"`, `"lzo"` or `"ucl"`. It always returns a tuple of 3 elements. When successful, the first element of this tuple has a positive value, and is 0 (zero) when library is not available (for example LZO or UCL). In case the library is available, the second element of tuple contains the library version and the third element the date (if available) of that version.

4.2 The `File` class

This class is returned when a `PyTables` file is opened with the `openFile` function. It has methods to flush and close files. Also, the `File` class offer methods to create, rename and delete nodes, as well as to traverse the object tree. One of its attributes (`rootUEP`) represents the *user entry point* to the object tree attached to the file.

Next, we will discuss the attributes and methods for File class[1].

4.2.1 `File` instance variables

filename Filename opened.

format_version The `PyTables` version number of this file.

isopen It takes the value 1 if the underlying file is open. 0 otherwise.

mode Mode in which the filename was opened.

root The *root* of the object tree hierarchy. It is a `Group` instance.

rootUEP The UEP (User Entry Point) group in file (see 4.1.2).

title The title of the root group in file.

trMap This is a dictionary that maps node names between python and HDF5 domain names. Its initial values are set from the *trMap* parameter passed to the `openFile` function. You can change its contents *after* a file is opened and the new map will take effect over any new object added to the tree.

[1] On the following, the term `Leaf` will whether refer to a `Table`, `Array`, `EArray`, `VLArray` or `UnImplemented` node object.

filters Container for filter properties associated to this file. See section 4.12.1 for more information on this object.

objects Dictionary with all objects (groups or leaves) on tree.

groups Dictionary with all object groups on tree.

leaves Dictionary with all object leaves on tree.

4.2.2 `File` methods

createGroup(where, name, title='', filters=None)

Create a new Group instance with name *name* in *where* location.

where The parent group where the new group will hang from. *where* parameter can be a path string (for example `"/level1/group5"`), or another Group instance.

name The name of the new group.

title A description for this group.

filters An instance of the `Filters` class (see section 4.12.1) that provides information about the desired I/O filters applicable to the leaves that hangs directly from this new group (unless other filters properties are specified for these leaves). Besides, if you do not specify filter properties for its child groups, they will inherit these ones.

createTable(where, name, description, title='', filters=None, expectedrows=10000)

Create a new `Table` instance with name *name* in *where* location.

where The parent group where the new table will hang from. *where* parameter can be a path string (for example `"/level1/leaf5"`), or Group instance.

name The name of the new table.

description An instance of a user-defined class (derived from the `IsDescription` class) where table fields are defined. However, in certain situations, it is more handy to allow this description to be supplied as a dictionary (for example, when you do not know beforehand which structure will have your table). In such a cases, you can pass the description as a dictionary as well. See section 3.3 for an example of use. Finally, a `RecArray` object from the `numarray` package is also accepted, and all the information about columns and other metadata is used as a basis to create the `Table` object. Moreover, if the `RecArray` has actual data this is also injected on the newly created `Table` object.

title A description for this object.

filters An instance of the `Filters` class (see section 4.12.1) that provides information about the desired I/O filters to be applied during the life of this object.

expectedrows An user estimate of the number of records that will be on table. If not provided, the default value is appropriate for tables until 1 MB in size (more or less, depending on the record size). If you plan to save bigger tables you should provide a guess; this will optimize the HDF5 B-Tree creation and management process time and memory used. See section 5.4 for a discussion on that issue.

createArray(where, name, object, title='')

Create a new `Array` instance with name *name* in *where* location.

object The regular array to be saved. Currently accepted values are: lists, tuples, scalars (int and float), strings and (multidimensional) `Numeric` and `NumArray` arrays (including `CharArrays` string arrays). However, these objects must be regular (i.e. they cannot be like, for example, `[[1,2],2]`). Also, objects that has some of its dimension equal to zero are not supported (this will be solved when unlimited arrays will be implemented).

See `createTable` description 4.2.2 for more information on the *where*, *name* and *title*, parameters.

createEArray(where, name, atom, title='', filters=None, expectedrows=1000)

Create a new `EArray` instance with name *name* in *where* location.

atom An `Atom` instance representing the *shape*, *type* and *flavor* of the atomic objects to be saved. One (and only one) of the shape dimensions **must** be 0. The dimension being 0 means that the resulting `EArray` object can be extended along it. Multiple enlargeable dimensions are not supported right now. See section 4.11.3 for the supported set of `Atom` class descendants.

expectedrows In the case of enlargeable arrays this represents an user estimate about the number of row elements that will be added to the growable dimension in the EArray object. If not provided, the default value is 1000 rows. If you plan to create both much smaller or much bigger EArrays try providing a guess; this will optimize the HDF5 B-Tree creation and management process time and the amount of memory used.

See `createTable` description 4.2.2 for more information on the *where*, *name*, *title*, and *filters* parameters.

createVLArray(where, name, atom=None, title='', filters=None, expectedsizeinMB=1.0)

Create a new `VLArray` instance with name *name* in *where* location. See the section 4.8 for a description of the `VLArray` class.

atom An `Atom` instance representing the shape, type and flavor of the atomic object to be saved. See section 4.11.3 for the supported set of `Atom` class descendants.

expectedsizeinMB An user estimate about the size (in MB) in the final `VLArray` object. If not provided, the default value is 1 MB. If you plan to create both much smaller or much bigger VLA's try providing a guess; this will optimize the HDF5 B-Tree creation and management process time and the amount of memory used.

See `createTable` description 4.2.2 for more information on the *where*, *name*, *title*, and *filters* parameters.

getNode(where, name='', classname='')

Returns the object node *name* under *where* location.

where Can be a path string or `Group` instance. If *where*

doesn't exists or has already a child called

name, a `ValueError` error is raised.

name The object name desired. If *name* is a null string (''), or not supplied, this method assumes to find the object in *where*.

classname If supplied, returns only an instance of this class name. Possible values are: `'Group'`, `'Leaf'`, `'Table'`, `'Array'`, `'EArray'`, `'VLArray'` and `'UnImplemented'`. Note that these values are strings.

getAttrNode(where, attrname, name='')

Returns the attribute *attrname* under *where.name* location.

where Can be a path string or `Group` instance. If *where* doesn't exists or has not a child called *name*, a `ValueError` error is raised.

attrname The name of the attribute to get.

name The node name desired. If *name* is a null string (''), or not supplied, this method assumes to find the object in *where*.

setAttrNode(where, attrname, attrvalue, name='')

Sets the attribute *attrname* with value *attrvalue* under *where.name* location.

where Can be a path string or `Group` instance. If *where* doesn't exists or has not a child called *name*, a `ValueError` error is raised.

attrname The name of the attribute to set on disk.

attrvalue The value of the attribute to set. Only strings attributes are supported natively right now. However, you can always use `(c)Pickle` so as to serialize any object you want save therein.

name The node name desired. If *name* is a null string (''), or not supplied, this method assumes to find the object in *where*.

listNodes(where, classname='')

Returns a list with all the object nodes (Group or Leaf) hanging from *where*. The list is alpha-numerically sorted by node name.

where The parent group. Can be a path string or `Group` instance.

classname If a *classname* parameter is supplied, the iterator will return only instances of this class (or subclasses of it). The only supported classes in *classname* are `'Group'`, `'Leaf'`, `'Table'`, `'Array'`, `'EArray'`, `'VLArray'` and `'UnImplemented'`. Note that these values are strings.

removeNode(where, name = "", recursive=0)

Removes the object node *name* under *where* location.

where Can be a path string or `Group` instance. If *where* doesn't exists or has not a child called *name*, a `LookupError` error is raised.

name The name of the node to be removed. If not provided, the *where* node is changed.

recursive If not supplied, the object will be removed only if it has no children. If supplied with a true value, the object and all its descendants will be completely removed.

renameNode(where, newname, name)

Rename the object node *name* under *where* location.

where Can be a path string or `Group` instance. If *where* doesn't exists or has not a child called *name*, a `LookupError` error is raised.

newname Is the new name to be assigned to the node.

name The name of the node to be changed. If not provided, the *where* node is changed.

walkGroups(where='/')

Iterator that returns the list of Groups (not Leaves) hanging from *where*. If *where* is not supplied, the root object is taken as origin. The returned Group list is in a top-bottom order, and alpha-numerically sorted when they are at the same level.

where The origin group. Can be a path string or `Group` instance.

flush()

Flush all the leaves in the object tree.

close()

Flush all the leaves in object tree and close the file.

4.2.3 `File` special methods

Following are described the methods that automatically trigger actions when a `File` instance is accessed in a special way (e.g., `fileh("/detector")` will cause a call to `group.__call__("/detector")`).

__call__(where="/", classname="")

Recursively iterate over the children in the `File` instance. It takes two parameters:

where If supplied, the iteration starts from this group.

classname *(String)* If supplied, only instances of this class are returned.

Example of use:

```
# Recursively print all the nodes hanging from '/detector'
print "Nodes hanging from group '/detector':"
for node in h5file("/detector"):
    print node
```

__iter__()

Iterate over the children on the `File` instance. However, this does not accept parameters. This iterator *is recursive*.

Example of use:

```
# Recursively list all the nodes in the object tree
h5file = tables.openFile("vlarray1.h5")
print "All nodes in the object tree:"
for node in h5file:
    print node
```

4.3 The `Group` class

Instances of this class are a grouping structure containing instances of zero or more groups or leaves, together with supporting metadata.

Working with groups and leaves is similar in many ways to working with directories and files, respectively, in a Unix filesystem. As with Unix directories and files, objects in the object tree are often described by giving their full (or absolute) path names. This full path can be specified either as a string (like in

`'/group1/group2'`) or as a complete object path written in *natural name* schema (like in `file.root.group1.group2`) as discussed in the section 1.2.

A collateral effect of the *natural naming* schema is that you must be aware when assigning a new attribute variable to a Group object to not collide with existing children node names. For this reason and to not pollute the children namespace, it is explicitly forbidden to assign "normal" attributes to Group instances, and the only ones allowed must start with some reserved prefixes, like "`_f_`" (for methods) or "`_v_`" (for instance variables) prefixes. Any attempt to assign a new attribute that does not starts with these prefixes, will raise a `NameError` exception.

Other effect is that you cannot use reserved Python names or other non-allowed python names (like for example "$a" or "44") as node names. You can, however, make use of the `trMap` (translation map dictionary) parameter in the `openFile` function (see section 4.1.2) in order to use non-valid Python names as node names in the file.

4.3.1 `Group` instance variables

_v_title A description for this group.

_v_name The name of this group.

_v_hdf5name The name of this group in HDF5 file namespace.

_v_pathname A string representation of the group location in tree.

_v_parent The parent Group instance.

_v_rootgroup Pointer to the root group object.

_v_file Pointer to the associated File object.

_v_depth The depth level in tree for this group.

_v_childs Dictionary with all nodes (groups or leaves) hanging from this instance.

_v_groups Dictionary with all node groups hanging from this instance.

_v_leaves Dictionary with all node leaves hanging from this instance.

_v_attrs The associated `AttributeSet` instance (see 4.10).

_v_filters Container for filter properties. See section 4.12.1 for more information on this object.

4.3.2 `Group` methods

This class define the `__setattr__`, `__getattr__` and `__delattr__` and they work as normally intended. So, you can access, assign or delete childs to a group by just using the next constructs:

```
# Add a Table child instance under group with name "tablename"
group.tablename = Table(recordDict, "Record instance")
table = group.tablename     # Get the table child instance
del group.tablename         # Delete the table child instance
```

Caveat: The following methods are documented for completeness, and they can be used without any problem. However, you should use the high-level counterpart methods in the `File` class, because these are most used in documentation and examples, and are a bit more powerful than those exposed here.

_f_join(name) Helper method to correctly concatenate a name child object with the pathname of this group.

_f_rename(newname) Change the name of this group to *newname*.

_f_remove(recursive=0) Remove this object. If *recursive* is true, force the removal even if this group has children.

_f_getAttr(attrname) Gets the HDF5 attribute *attrname* of this group.

_f_setAttr(attrname, attrvalue) Sets the attribute *attrname* of this group to the value *attrvalue*. Only string values are allowed.

_f_listNodes(classname='') Returns a *list* with all the object nodes hanging from this instance. The list is alpha-numerically sorted by node name. If a *classname* parameter is supplied, it will only return instances of this class (or subclasses of it). The supported classes in *classname* are `'Group'`, `'Leaf'`, `'Table'` and `'Array'`, `'EArray'`, `'VLArray'` and `'UnImplemented'`.

_f_walkGroups() Iterator that returns the list of Groups (not Leaves) hanging from *self*. The returned Group list is in a top-bottom order, and alpha-numerically sorted when they are at the same level.

_f_close() Close this group, making it and its children unaccessible in the object tree.

4.3.3 `Group` special methods

Following are described the methods that automatically trigger actions when a `Group` instance is accessed in a special way (e.g., `group("Table")` will be equivalent to a call to `group.__call__("Table")`).

__call__(classname="", recursive=0)

Iterate over the childs in the `Group` instance. It takes two parameters:

classname *(String)* If supplied, only instances of this class are returned.

recursive *(Integer)* If false, only childs hanging immediately after the group are returned. If true, a recursion over all the groups hanging from it is performed.

Example of use:

```
# Recursively print all the arrays hanging from '/'
print "Arrays the object tree '/':"
for array in h5file.root(classname="Array", recursive=1):
    print array
```

__iter__()

Iterate over the childs on the group instance. However, this does not accept parameters. This iterator is **not** recursive.

Example of use:

```
# Non-recursively list all the nodes hanging from '/detector'
print "Nodes in '/detector' group:"
for node in h5file.root.detector:
    print node
```

4.4 The `Leaf` class

The goal of this class is to provide a place to put common functionality of all its descendants as well as provide a way to help classifying objects on the tree. A `Leaf` object is an end-node, that is, a node that can hang directly from a group object, but that is not a group itself and, thus, it cannot have descendents. Right now, the set of end-nodes is composed by `Table`, `Array`, `EArray`, `VLArray` and `UnImplemented` class instances. In fact, all the previous classes inherits from the `Leaf` class.

4.4.1 `Leaf` instance variables

The public variables and methods that class descendants inherits from `Leaf` are listed below.

name The Leaf node name in Python namespace.

hdf5name The Leaf node name in HDF5 namespace.

objectID The HDF5 object ID of the Leaf node.

title The Leaf title (actually a property rather than a plain attribute).

shape The shape of the associated data in the Leaf.

byteorder The byteorder of the associated data of the Leaf.

attrs The associated `AttributeSet` instance (see 4.10).

filters Container for filter properties. See section 4.12.1 for more information on this object.

Besides, the next instance variables are also defined and have similar meaning as its counterparts in the `Group` class:

_v_hdf5name The name of this leaf in HDF5 file namespace.

_v_pathname A string representation of the leaf location in tree.

_v_parent The parent `Group` instance.

_v_rootgroup Pointer to the root `Group` object.

_v_file Pointer to the associated `File` object.

_v_depth The depth level in tree for this leaf.

4.4.2 `Leaf` methods

copy(where, name, title=None, filters=None, copyuserattrs=1, start=0, stop=None, step=1)

Copy this leaf into another location. The meaning of the parameters is explained below:

where Can be a path string or `Group` instance. If *where* doesn't exists or has not a child called *name*, a `LookupError` error is raised.

name The name of the destination node.

title The new title for destination. If None, the original title is copied.

filters An instance of the `Filters` (see section 4.12.1) class. A None value means that the source properties are copied *as is*.

copyuserattrs Whether copy the user attributes of the source leaf to the destination or not. The default is copying them.

The meaning of the *start*, *stop* and *step* parameters is the same as in the `range()` python function, except that negative values of `step` are not allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither *start* nor *stop*, then **all the rows** in the object are selected.

remove()

Remove this leaf.

rename(newname)

Change the name of this leaf to *newname*.

getAttr(attrname)

Gets the HDF5 attribute *attrname* of this leaf.

setAttr(attrname, attrvalue)

Sets the attribute *attrname* of this leaf to the value *attrvalue*.

flush()

Flush the leaf buffers (if any).

close()

Flush the leaf buffers (if any) and close the dataset.

4.5 The `Table` class

Instances of this class represents table objects in the object tree. It provides methods to read/write data and from/to table objects in the file.

Data can be read from or written to tables by accessing to an special object that hangs from `Table`. This object is an instance of the `Row` class (see 4.5.4). See the tutorial sections chapter 3 on how to use the `Row` interface. The columns of the tables can also be easily accessed (and more specifically, they can be read but not written) by making use of the `Column` class, through the use of an *extension* of the natural naming schema applied inside the tables. See the section 4.5.6 for some examples of use of this capability.

Note that this object inherits all the public attributes and methods that `Leaf` already has.

4.5.1 `Table` instance variables

description The metaobject describing this table.

row The `Row` instance for this table (see 4.5.4).

nrows The number of rows in this table.

rowsize The size, in bytes, of each row.

cols A `Cols` (see section 4.5.5) instance that serves as accessor to `Column` (see section 4.5.6) objects.

colnames The field names for the table (list).

coltypes The data types for the table fields (dictionary).

colshapes The shapes for the table fields (dictionary).

4.5.2 `Table` methods

append(rows=None)

Append a series of rows to this `Table` instance. *rows* is an object that can keep the rows to be append in several formats, like a `RecArray`, a list of tuples, list of `Numeric`/`NumArray`/`CharArray` objects, string, Python buffer or None (no append will result). Of course, this *rows* object has to be compliant with the underlying format of the `Table` instance or a `ValueError` will be issued.

Example of use:

```
from tables import *
class Particle(IsDescription):
    name        = StringCol(16, pos=1)   # 16-character String
    lati        = IntCol(pos=2)        # integer
    longi       = IntCol(pos=3)        # integer
    pressure    = Float32Col(pos=4)    # float  (single-precision)
    temperature = FloatCol(pos=5)      # double (double-precision)

fileh = openFile("test4.h5", mode = "w")
table = fileh.createTable(fileh.root, 'table', Particle, "A table")
# Append several rows in only one call
table.append([("Particle:     10", 10, 0, 10*10, 10**2),
              ("Particle:     11", 11, -1, 11*11, 11**2),
              ("Particle:     12", 12, -2, 12*12, 12**2)])
fileh.close()
```

iterrows(start=None, stop=None, step=1)

Returns an iterator yielding `Row` (see section 4.5.4) instances built from rows in table. If a range is supplied (i.e. some of the *start*, *stop* or *step* parameters are passed), only the appropriate rows are returned. Else, all the rows are returned. See also the `__call__()` and `__iter__()` special methods in section 4.5.3 for shorter ways to call this iterator.

The meaning of the *start*, *stop* and *step* parameters is the same as in the `range()` python function, except that negative values of `step` are not allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither *start* nor *stop*, then **all the rows** in the object are selected.

read(start=None, stop=None, step=1, field=None, flavor="numarray")

Returns the actual data in `Table`. If *field* is not supplied, it returns the data as a `RecArray` object table.

The meaning of the *start*, *stop* and *step* parameters is the same as in the `range()` python function, except that negative values of `step` are not allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

The rest of the parameters are described next:

field If specified, only the column *field* is returned as a `NumArray` object. If this is not supplied, all the fields are selected and a `RecArray` is returned.

flavor When a field in table is selected, passing a *flavor* parameter make an additional conversion to happen in the default `"numarray"` returned object. *flavor* must have any of the next values: `"numarray"` (i.e. no conversion is made), `"Numeric"`, `"Tuple"` or `"List"`.

removeRows(start=None, stop=None)

Removes a range of rows in the table. If only *start* is supplied, this row is to be deleted. If a range is supplied, i.e. both the *start* and *stop* parameters are passed, all the rows in the range are removed[2]. A *step* parameter is not supported yet.

start Sets the starting row to be removed. It accepts negative values meaning that the count starts from the end. A value of 0 means the first row.

stop Sets the last row to be removed to *stop* - 1, i.e. the end point is omitted (in the Python `range` tradition). It accepts, likewise *start*, negative values. A special value of `None` means the last row.

[2] However, for `removeRows()` to work, you need that the rows **after** the `stop` parameter will fit in-memory so as to method to work. This limitation will be hopefully removed in a future version.

4.5.3 `Table` special methods

Following are described the methods that automatically trigger actions when a `Table` instance is accessed in a special way (e.g., `table["var2"]` will be equivalent to a call to `table.__getitem__("var2")`).

__call__(start=None, stop=None, step=1)

It returns the same iterator than `Table.iterrows(start, stop, step)`. It is, therefore, a shorter way to call it.

Example of use:

```
result = [ row['var2'] for row in table(step=4)
                            if row['var1'] <= 20 ]
```

Which is equivalent to:

```
result = [ row['var2'] for row in table.iterrows(step=4)
                            if row['var1'] <= 20 ]
```

__iter__()

It returns the same iterator than `Table.iterrows(0,0,1)`. However, this does not accept parameters.

Example of use:

```
result = [ row['var2'] for row in table
                                if row['var1'] <= 20 ]
```

Which is equivalent to:

```
result = [ row['var2'] for row in table.iterrows()
                                if row['var1'] <= 20 ]
```

__getitem__(key)

It takes different actions depending on the type of the `key` parameter:

`key` is an `Integer` The corresponding table row is returned as a `RecArray.Record` object.

`key` is a `Slice` The row slice determined by `key` is returned as a `RecArray` object.

`key` is a `String` The `key` is interpreted as a *column* name of the table, and, if it exists, it is read and returned as a `NumArray` or `CharArray` object (whatever is appropriate).

Example of use:

```
record = table[4]
recarray = table[4:1000:2]
narray = table["var2"]
```

Which is equivalent to:

```
record = table.read(start=4)[0]
recarray = table.read(start=4, stop=1000, step=2)
narray = table.read(field="var2")
```

4.5.4 The Row class

This class is used to fetch and set values on the table fields. It works very much like a dictionary, where the keys are the field names of the associated table and the values are the values of those fields in a specific row.

This object turns out to actually be an extension type, so you won't be able to access their documentation interactively. Neither you won't be able to access it's internal attributes (they are not directly accessible from Python), although that *accessors* (i.e. methods that return an internal attribute) has been defined for the most important variables.

Row methods

append() Once you have filled the proper fields for the current row, calling this method actually commit this data to the disk (actually data is written to the output buffer).

nrow() Accessor that returns the current row in the table. It is useful to know which row is being dealt with in the middle of a loop.

4.5.5 The Cols class

This class is used as an *accessor* to the table columns following the natural name convention, so that you can access the different columns because there exist one attribute with the name of the columns for each associated `Column` instances. Besides, and like the `Row` class, it works similar to a dictionary, where the keys are the column names of the associated table and the values are `Column` instances. See section 4.5.6 for examples of use.

4.5.6 The Column class

Each instance of this class is associated with one column of every table. These instances are used to fetch (but not set) actual data from the table columns. The access interface is like a regular list, and you can select individual values or slices.

Column instance variables

table The parent `Table` instance.

name The name of the associated column.

Column methods

__getitem__(key) Returns a column element or slice. It takes different actions depending on the type of the *key* parameter: If *key* is an integer, the corresponding element in the column is returned as a scalar object or as a `NumArray`/`CharArray` object, depending on its shape. If *key* is a slice, the row range determined by this slice is returned as a `NumArray` or `CharArray` object (whichever is appropriate).

Example of use:

```
print "Column handlers:"
for name in table.colnames:
    print table.cols[name]
print
print "Some selections:"
print "Select table.cols.name[1]-->", table.cols.name[1]
print "Select table.cols.name[1:2]-->", table.cols.name[1:2]
print "Select table.cols.lati[1:3]-->", table.cols.lati[1:3]
print "Select table.cols.pressure[:]-->", table.cols.pressure[:]
print "Select table.cols['temperature'][:]-->", table.cols['temperature'][:]
```

and the output of this for a certain arbitrary table is:

```
Column handlers:
/table.cols.name (Column(1,), CharType)
/table.cols.lati (Column(2,), Int32)
/table.cols.longi (Column(1,), Int32)
/table.cols.pressure (Column(1,), Float32)
/table.cols.temperature (Column(1,), Float64)

Some selections:
Select table.cols.name[1]--> Particle:     11
Select table.cols.name[1:2]--> ['Particle:     11']
Select table.cols.lati[1:3]--> [[11 12]
 [12 13]]
Select table.cols.pressure[:]--> [  90.  110.  132.]
Select table.cols['temperature'][:]--> [ 100.  121.  144.]
```

See the `examples/table2.py` for a more complete example.

4.6 The `Array` class

Represents an array on file. It provides methods to write/read data to/from array objects in the file. This class does not allow to enlarge the datasets on disk; see the `EArray` descendant in section 4.7 if you want enlargeable dataset support and/or compression features.

Caveat: All `Numeric` and `numarray` data types are supported except those that corresponds to complex data types[3]. See `numarray` manual (Greenfield *et al.*) to know more about the supported data types, or see appendix A.

Note that this object inherits all the public attributes and methods from `Leaf` already provides.

4.6.1 `Array` instance variables

flavor The object representation for this array. It can be any of *"NumArray"*, *"CharArray"* *"Numeric"*, *"List"*, *"Tuple"*, *"String"*, *"Int"* or *"Float"* values.

nrows The length of the first dimension of Array.

nrow On iterators, this is the index of the current row.

type The type class of the represented array.

itemsize The size of the base items. Specially useful for `CharArray` objects.

4.6.2 `Array` methods

Note that, as this object has no internal I/O buffers, it is not necessary to flush() method inherited from `Leaf`.

iterrows(start=None, stop=None, step=1)

Returns an iterator yielding `numarray` instances built from rows in array. The return rows are taken from the first dimension in case of an `Array` instance and the enlargeable dimension in case of an `EArray` instance. If a range is supplied (i.e. some of the *start*, *stop* or *step* parameters are passed), only the appropriate rows are returned. Else, all the rows are returned. See also the `__call__()` and `__iter__()` special methods in section 4.6.3 for shorter ways to call this iterator.

[3] However, these might be included in the future

The meaning of the *start*, *stop* and *step* parameters is the same as in the `range()` python function, except that negative values of `step` are not allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

read(start=None, stop=None, step=1)

Read the array from disk and return it as a `numarray` (default) object, or an object with the same original *flavor* that it was saved. It accepts start, stop and step parameters to select rows (the first dimension in the case of an `Array` instance and the *enlargeable* dimension in case of an `EArray`) for reading.

The meaning of the *start*, *stop* and *step* parameters is the same as in the `range()` python function, except that negative values of `step` are not allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

4.6.3 `Array` special methods

Following are described the methods that automatically trigger actions when an `Array` instance is accessed in a special way (e.g., `array[2:3,...,::2]` will be equivalent to a call to `array.__getitem__(slice(2,3, None), Ellipsis, slice(None, None, 2))`).

__call__(start=None, stop=None, step=1)

It returns the same iterator than `Array.iterrows(start, stop, step)`. It is, therefore, a shorter way to call it.

Example of use:

```
result = [ row for row in arrayInstance(step=4) ]
```

Which is equivalent to:

```
result = [ row for row in arrayInstance.iterrows(step=4) ]
```

__iter__()

It returns the same iterator than `Array.iterrows(0,0,1)`. However, this does not accept parameters.

Example of use:

```
result = [ row[2] for row in array ]
```

Which is equivalent to:

```
result = [ row[2] for row in array.iterrows(0, 0, 1) ]
```

__getitem__(key)

It returns a `numarray` (default) object (or an object with the same original *flavor* that it was saved) containing the slice of rows stated in the `key` parameter. The set of allowed tokens in `key` is the same as extended slicing in python (the `Ellipsis` token included).

Example of use:

```
array1 = array[4]   # array1.shape == array.shape[1:]
array2 = array[4:1000:2]  # len(array2.shape) == len(array.shape)
array3 = array[::2, 1:4, :]
array4 = array[1, ..., ::2, 1:4, 4:] # General slice selection
```

4.7 The `EArray` class

This is a child of the `Array` class (see 4.6) and as such, `EArray` represents an array on the file. The difference is that `EArray` allows to enlarge datasets along any single dimension[4] you select. Another important difference is that it also support compression.

So, in addition to the attributes and methods that `EArray` inherits from `Array`, it supports a few more that provides a way to enlarge the arrays on disk. Following are described the new variables and methods as well as some that already exists in `Array` but that differ somewhat on the meaning and/or functionality in the `EArray` context.

4.7.1 `EArray` instance variables

atom The class instance choosed for the atom object (see section 4.11.3).

extdim The enlargeable dimension.

nrows The length of the enlargeable dimension.

4.7.2 `EArray` methods

append(object)

Appends an `object` to the underlying dataset. Obviously, this object has to have the same type as the `EArray` instance, and if not, a

`TypeError` is issued. In the same way, the

dimensions of the `object` has to conform those of `EArray`, that is, all the dimensions has to be the same except, of course, that of the enlargeable dimension which can be of any length (even 0!).

Example of use (code available in `examples/earray1.py`):

```
import tables
from numarray import strings

fileh = tables.openFile("earray1.h5", mode = "w")
a = tables.StringAtom(shape=(0,), length=8)
# Use ’a’ as the object type for the enlargeable array
array_c = fileh.createEArray(fileh.root, ’array_c’, a, "Chars")
array_c.append(strings.array([’a’*2, ’b’*4], itemsize=8))
array_c.append(strings.array([’a’*6, ’b’*8, ’c’*10], itemsize=8))

# Read the string EArray we have created on disk
for s in array_c:
    print "array_c[%s] => ’%s’" % (array_c.nrow, s)
# Close the file
fileh.close()
```

and the output is:

```
array_c[0] => ’aa’
array_c[1] => ’bbbb’
array_c[2] => ’aaaaaa’
array_c[3] => ’bbbbbbbb’
array_c[4] => ’cccccccc’
```

[4] In the future, multiple enlargeable dimensions might be implemented as well.

4.8 The VLArray class

Instances of this class represents array objects in the object tree with the property that their rows can have a **variable** number of (homogeneous) elements (called *atomic* objects, or just *atoms*). Variable length arrays (or *VLA's* for short), similarly to `Table` instances, can have only one dimension, and likewise `Table`, the compound elements (the *atoms*) of the rows of `VLArrays` can be fully multidimensional objects.

`VLArray` provides methods to read/write data from/to variable length array objects residents on disk. Also, note that this object inherits all the public attributes and methods that `Leaf` already has.

4.8.1 VLArray instance variables

atom The class instance choosed for the atom object (see section 4.11.3).

nrow On iterators, this is the index of the current row.

nrows The total number of rows.

4.8.2 VLArray methods

append(object1, object2, ...)

Append the `objects` passed as parameters to a single row in the `VLArray` instance. The type of the objects has to be compliant with the `VLArray.atom` instance type.

Example of use (code available in `examples/vlarray1.py`):

```
import tables
from Numeric import *   # or, from numarray import *

# Create a VLArray:
fileh = tables.openFile("vlarray1.h5", mode = "w")
vlarray = fileh.createVLArray(fileh.root, 'vlarray1',
tables.Int32Atom(flavor="Numeric"),
                 "ragged array of ints", Filters(complevel=1))
# Append some (variable length) rows
# All these different parameter specification are accepted:
vlarray.append(array([5, 6]))
vlarray.append(array([5, 6, 7]))
vlarray.append([5, 6, 9, 8])
vlarray.append(5, 6, 9, 10, 12)

# Now, read it through an iterator
for x in vlarray:
    print vlarray.name+"["+str(vlarray.nrow)+"]-->", x

# Close the file
fileh.close()
```

And the output for this looks like:

```
vlarray1[0]--> [5 6]
vlarray1[1]--> [5 6 7]
vlarray1[2]--> [5 6 9 8]
vlarray1[3]--> [ 5  6  9 10 12]
```

iterrows(start=None, stop=None, step=1)

Returns an iterator yielding one row per iteration. If a range is supplied (i.e. some of the *start*, *stop* or *step* parameters are passed), only the appropriate rows are returned. Else, all the rows are returned. See also the `__call__()` and `__iter__()` special methods in section 4.8.3 for shorter ways to call this iterator.

The meaning of the *start*, *stop* and *step* parameters is the same as in the `range()` python function, except that negative values of `step` are not allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

read(start=None, stop=None, step=1)

Returns the actual data in `VLArray`. As the lengths of the different rows are variable, the returned value is a python list, with as many entries as specified rows in the range parameters.

The meaning of the *start*, *stop* and *step* parameters is the same as in the `range()` python function, except that negative values of `step` are not allowed. Moreover, if only `start` is specified, then `stop` will be set to `start+1`. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

4.8.3 `VLArray` special methods

Following are described the methods that automatically trigger actions when a `VLArray` instance is accessed in a special way (e.g., `vlarray[2:5]` will be equivalent to a call to `vlarray.__getitem__(slice(2,5,None))`).

__call__(start=None, stop=None, step=1)

It returns the same iterator than `VLArray.iterrows(start, stop, step)`. It is, therefore, a shorter way to call it.

Example of use:

```
for row in vlarray(step=4):
    print vlarray.name+"["+str(vlarray.nrow)+"]-->", row
```

Which is equivalent to:

```
for row in vlarray.iterrows(step=4):
    print vlarray.name+"["+str(vlarray.nrow)+"]-->", row
```

__iter__()

It returns the same iterator than `VLArray.iterrows(0,0,1)`. However, this does not accept parameters.

Example of use:

```
result = [ row for row in vlarray ]
```

Which is equivalent to:

```
result = [ row for row in vlarray.iterrows() ]
```

__getitem__(key)

It returns the slice of rows determined by `key`, which can be an integer index or an extended slice. The returned value is a list of objects of type `array.atom.type`.

Example of use:

```
list1 = vlarray[4]
list2 = vlarray[4:1000:2]
```

4.9 The `UnImplemented` class

Instances of this class represents an unimplemented dataset in a generic HDF5 file. When reading such a file (i.e. one that has not been created with `PyTables`, but with some other HDF5 library based tool), chances are that the specific combination of *datatypes* and/or *dataspaces* in some dataset might not be supported by `PyTables` yet. In such a case, this dataset will be mapped into the `UnImplemented` class and hence, the user will still be able to build the complete object tree of this generic HDF5 file, as well as enabling the access (both read and *write*) of the attributes of this dataset and some metadata. Of course, the user won't be able to read the actual data on it.

This is an elegant way to allow users to work with generic HDF5 files despite the fact that some of its datasets would not be supported by `PyTables`. However, if you are really interested in having access to an unimplemented dataset, please, get in contact with the developer team.

This class does not have any public instance variables, except those inherited from the `Leaf` class (see 4.4).

4.10 The `AttributeSet` class

Represents the set of attributes of a node (Leaf or Group). It provides methods to create new attributes, open, rename or delete existing ones.

Like in `Group` instances, `AttributeSet` instances make use of the *natural naming* convention, i.e. you can access the attributes on disk like if they were *normal* `AttributeSet` attributes. This offers the user a very convenient way to access (but also to set and delete) node attributes by simply specifying them like a *normal* attribute class.

Caveat: All Python data types are supported. The scalar ones (i.e. String, Int and Float) are mapped directly to the HDF5 counterparts, so you can correctly visualize them with any HDF5 tool. However, the rest of the data types and more general objects are serialized using `cPickle`, so you will be able to correctly retrieve them only from a Python-aware HDF5 library. Hopefully, the list of supported native attributes will be extended to fully multidimensional arrays sometime in the future.

4.10.1 `AttributeSet` instance variables

_v_node The parent node instance.

_v_attrnames List with all attribute names.

_v_attrnamessys List with system attribute names.

_v_attrnamesuser List with user attribute names.

4.10.2 `AttributeSet` methods

Note that this class define the `__setattr__`, `__getattr__` and `__delattr__` and they work as normally intended. So, you can access, assign or delete attributes on disk by just using the next constructs:

```
leaf.attrs.myattr = "string attr"  # Set the attribute myattr
attrib = leaf.attrs.myattr  # Get the attribute myattr
del leaf.attrs.myattr       # Delete the attribute myattr
```

_f_list(attrset = "user") Return the list of attributes of the parent node.

> **attrset** Selects the attribute set to be returned. An `"user"` value returns only the user attributes. This is the default. `"sys"` returns only the system (some of which are read-only) attributes. `"readonly"` returns the system read-only attributes. `"all"` returns both the system and user attributes.

_f_rename(oldattrname, newattrname) Rename an attribute.

4.11 Declarative classes

In this section a series of classes that are meant to *declare* datatypes that are required for primary `PyTables` (like `Table` or `VLArray`) objects are described.

4.11.1 The `IsDescription` class

This class is in fact a so-called *metaclass* object. There is nothing special on this fact, except that their subclasses attributes are transformed during its instantiation phase, and new methods for instances are defined based on the values of the class attributes.

It is designed to be used as an easy, yet meaningful way to describe the properties of `Table` objects through the use of classes that inherit properties from it. In order to define such a special class, you have to declare it as descendant of *IsDescription*, with many attributes as columns you want in your table. The name of these attributes will become the name of the columns, while its values are the properties of the columns that are obtained through the use of the `Col` class constructor. See the section 4.11.2 for instructions on how define the properties of the table columns.

Then, you can pass an instance of this object to the `Table` constructor, where all the information it contains will be used to define the table structure. See the section 3.3 for an example on how that works.

4.11.2 The `Col` class and its descendants

The `Col` class is used as a mean to declare the different properties of a table column. In addition, a series of descendant classes are offered in order to make these column descriptions easier to the user. In general, it is recommended to use these descendant classes, as they are more meaningful when found in the middle of the code.

Note that the only public method accessible in these classes is the constructor itself.

Col(dtype="Float64", shape=1, dflt=None, pos=None) Declare the properties of a `Table` column.

- **dtype** The data type for the column. See the appendix A for a relation of data types supported in a `IsDescription` class declaration. The type description is accepted both in string format and as numarray data type.
- **shape** An integer or a tuple, that specifies the number of *dtype* items for each element (or shape, for multidimensional elements) of this column. For `CharType`

  columns, the last dimension is used as the length

  of the character strings. However, for this kind of objects, the use of `StringCol` subclass is strongly recommended.
- **dflt** The default value for elements of this column. If the user does not supply a value for an element while filling a table, this default value will be written to disk. If the user supplies an scalar value for a multidimensional column, this value is automatically *broadcasted* to all the elements in the column cell. If *dflt* is not supplied, an appropriate zero value (or *null* string) will be chosen by default.
- **pos** By default, columns are arranged in memory following an alpha-numerical order of the column names. In some situations, however, it is convenient to impose a user defined ordering. *pos* parameter allows the user to force the desired ordering.

StringCol(length=None, dflt=None, shape=1, pos=None) Declare a column to be of type `CharType`. The *length* parameter sets the length of the strings. The meaning of the other parameters are like in the `Col` class.

BoolCol(dflt=0, shape=1, pos=None) Define a column to be of type `Bool`. The meaning of the parameters are the same of those in the `Col` class.

IntCol(dflt=0, shape=1, itemsize=4, sign=1, pos=None) Declare a column to be of type `IntXX`, depending on the value of *itemsize* parameter, that sets the number of bytes of the integers in the column. *sign* determines whether the integers are signed or not. The meaning of the other parameters are the same of those in the `Col` class.

This class has several descendants:

Int8Col(dflt=0, shape=1, pos=None) Define a column of type `Int8`.

UInt8Col(dflt=0, shape=1, pos=None) Define a column of type `UInt8`.

Int16Col(dflt=0, shape=1, pos=None) Define a column of type `Int16`.

UInt16Col(dflt=0, shape=1, pos=None) Define a column of type `UInt16`.

Int32Col(dflt=0, shape=1, pos=None) Define a column of type `Int32`.

UInt32Col(dflt=0, shape=1, pos=None) Define a column of type `UInt32`.

Int64Col(dflt=0, shape=1, pos=None) Define a column of type `Int64`.

UInt64Col(dflt=0, shape=1, pos=None) Define a column of type `UInt64`.

FloatCol(dflt=0, shape=1, itemsize=8, pos=None) Define a column to be of type `FloatXX`, depending on the value of `itemsize`. The `itemsize` parameter sets the number of bytes of the floats in the column and the default is 8 bytes (double precision). The meaning of the other parameters are the same as those in the `Col` class.

This class has two descendants:

Float32Col(dflt=0.0, shape=1, pos=None) Define a column of type `Float32`.

Float64Col(dflt=0.0, shape=1, pos=None) Define a column of type `Float64`.

4.11.3 The `Atom` class and its descendants.

The `Atom` class is meant to declare the different properties of the *base element* (also known as *atom*) of `EArray` and `VLArray` objects. The `Atom` instances have the property that its length is always the same. However, you can grow objects along the extendable dimension in the case of `EArray` or put a variable number of them on a `VLArray` row. Moreover, the atoms are not restricted to scalar values, and they can be fully multidimensional objects.

A series of descendant classes are offered in order to make the use of these element descriptions easier. In general, it is recommended to use these descendant classes, as they are more meaningful when found in the middle of the code. Note that the only public methods accessible in these classes are the `atomsize()` method and the constructor itself. The `atomsize()` method returns the total length, in bytes, of the element base atom.

A description of the different constructors with their parameters follows:

Atom(dtype="Float64", shape=1, flavor="NumArray") Define properties for the base elements of `EArray` and `VLArray` objects.

dtype The data type for the base element. See the appendix A for a relation of data types supported. The type description is accepted both in string format and as numarray data type.

shape In a `EArray` context, it is a **tuple** specifing the shape of the object, and one (and only one) of its dimensions **must** be 0, meaning that the `EArray` object will be enlarged along this axis. In the case of a `VLArray`, it can be an integer with a value of 1 (one) or a tuple, that specifies whether the atom is an scalar (in the case of a 1) or has multiple dimensions (in the case of a tuple). For `CharType` elements, the last dimension is used as the length of the character strings. However, for this kind of objects, the use of `StringAtom` subclass is strongly recommended.

flavor The object representation for this atom. It can be any of *"CharArray"* or *"String"* for the `CharType` type and *"NumArray"*, *"Numeric"*, *"List"* or *"Tuple"* for the rest of the types. If the specified values differs from *CharArray* or *NumArray* values, the read atoms will be converted to that specific flavor. If not specified, the atoms will remain in their native format (i.e. `CharArray` or `NumArray`).

StringAtom(shape=1, length=None, flavor="CharArray") Define an atom to be of `CharType` type. The meaning of the *shape* parameter is the same as in the `Atom` class. *length* sets the length of the strings atoms. *flavor* can be whether `"CharArray"` or `"String"`. Unicode strings are not supported by this type; see the `VLStringAtom` class if you want Unicode support (only available for `VLAtom` objects).

BoolAtom(shape=1, flavor="NumArray") Define an atom to be of type `Bool`. The meaning of the parameters are the same of those in the `Atom` class.

IntAtom(shape=1, itemsize=4, sign=1, flavor="NumArray") Define an atom to be of type `IntXX`, depending on the value of *itemsize* parameter, that sets the number of bytes of the integers that conform the atom. *sign* determines whether the integers are signed or not. The meaning of the other parameters are the same of those in the `Atom` class.

This class has several descendants:

Int8Atom(shape=1, flavor="NumArray") Define an atom of type `Int8`.

UInt8Atom(shape=1, flavor="NumArray") Define an atom of type `UInt8`.

Int16Atom(shape=1, flavor="NumArray") Define an atom of type `Int16`.

UInt16Atom(shape=1, flavor="NumArray") Define an atom of type `UInt16`.

Int32Atom(shape=1, flavor="NumArray") Define an atom of type `Int32`.

UInt32Atom(shape=1, flavor="NumArray") Define an atom of type `UInt32`.

Int64Atom(shape=1, flavor="NumArray") Define an atom of type `Int64`.

UInt64Atom(shape=1, flavor="NumArray") Define an atom of type `UInt64`.

FloatAtom(shape=1, itemsize=8, flavor="NumArray") Define an atom to be of `FloatXX` type, depending on the value of `itemsize`. The `itemsize` parameter sets the number of bytes of the floats in the atom and the default is 8 bytes (double precision). The meaning of the other parameters are the same as those in the `Atom` class.

This class has two descendants:

Float32Atom(shape=1, flavor="NumArray") Define an atom of type `Float32`.

Float64Atom(shape=1, flavor="NumArray") Define an atom of type `Float64`.

Now, there come two special classes, `ObjectAtom` and `VLString`, that actually do not descend from `Atom`, but which goal is so similar that they should be described here. The difference between them and the `Atom` and descendents classes is that these special classes does not allow multidimensional atoms, nor multiple values per row. A *flavor* can't be specified neither as it is immutable (see below).

Caveat emptor: You are only allowed to use these classes to create `VLArray` objects, not `EArray` objects.

ObjectAtom() This class is meant to fit *any* kind of object in a row of an `VLArray` instance by using `cPickle` behind the scenes. Due to the fact that you cannot foresee how long will be the output of the `cPickle` serialization (i.e. the atom already has a *variable* length), you can only fit a representant of it per row. However, you can still pass several parameters to the `VLArray.append()` method as they will be regarded as a *tuple* of compound objects (the parameters), so that we still have only one object to be saved in a single row. It does not accept parameters and its flavor is automatically set to `"Object"`, so the reads of rows always returns an arbitrary python object. You can regard `ObjectAtom` types as an easy way to save an arbitrary number of generic python objects in a `VLArray` object.

VLStringAtom() This class describes a *row* of the `VLArray` class, rather than an *atom*. It differs from the `StringAtom` class in that you can only add one instance of it to one specific row, i.e. the `VLArray.append()` method only accepts one object when the base atom is of this type. Besides, it supports Unicode strings (contrarily to `StringAtom`) because it uses the UTF-8 codification (this is why its `atomsize()` method returns always 1) when serializing to disk. It does not accept any parameter and because its *flavor* is automatically set to `"VLString"`, the reads of rows always returns a python string. See the appendix B.3.4 if you are curious on how this is implemented at the low-level. You can regard `VLStringAtom` types as an easy way to save generic variable length strings.

See `examples/vlarray1.py` and `examples/vlarray2.py` for further examples on `VLArray`s, including object serialization and Unicode string management.

4.12 Helper classes

In this section are listed classes that does not fit in any other section and that mainly serves for ancillary purposes.

4.12.1 The `Filters` class

This class is meant to serve as a container that keeps information about the filter properties associated with the enlargeable leafs, that is `Table`, `EArray` and `VLArray`.

The public variables of `Filters` are listed below:

complevel The compression level (0 means no compression).

complib The compression filter used (in case of compressed dataset).

shuffle Whether the shuffle filter is active or not.

fletcher32 Whether the fletcher32 filter is active or not.

There are no `Filters` public methods with the exception of the constructor itself that is described next.

`Filters(complevel=0, complib="zlib", shuffle=1, fletcher32=0)`

The parameters that can be passed to the `Filters` class constructor are:

complevel Specifies a compress level for data. The allowed range is 0-9. A value of 0 disables compression. The default is that compression is disabled, that balances between compression effort and CPU consumption.

complib Specifies the compression library to be used. Right now, `"zlib"` (default), `"lzo"` and `"ucl"` values are supported. See section 5.2 for some advice on which library is better suited to your needs.

shuffle Whether or not to use the *shuffle* filter present in the `HDF5` library. This is normally used to improve the compression ratio (at the cost of consuming a little bit more CPU time). A value of 0 disables shuffling and 1 makes it active. The default value depends on whether compression is enabled or not; if compression is enabled, shuffling defaults to be active, else shuffling is disabled.

fletcher32 Whether or not to use the *fletcher32* filter in the HDF5 library. This is used to add a checksum on each data chunk. A value of 0 disables the checksum and it is the default.

Of course, you can also create an instance and then assign the ones you want to change. For example:

```
import numarray as na
from tables import *

fileh = openFile("test5.h5", mode = "w")
atom = Float32Atom(shape=(0,2))
filters = Filters(complevel=1, complib = "lzo")
filters.fletcher32 = 1
arr = fileh.createEArray(fileh.root, 'earray', atom, "A growable array",
```

```
                          filters = filters)
# Append several rows in only one call
arr.append(na.array([[1., 2.],
                     [2., 3.],
                     [3., 4.]], type=na.Float32))

# Print information on that enlargeable array
print "Result Array:"
print repr(arr)

fileh.close()
```

This enforces the use of the `LZO` library, a compression level of 1 and a fletcher32 checksum filter as well. See the output of this example:

```
Result Array:
/earray (EArray(3L, 2), fletcher32, shuffle, lzo(1)) 'A growable array'
  type = Float32
  shape = (3L, 2)
  itemsize = 4
  nrows = 3
  extdim = 0
  flavor = 'NumArray'
  byteorder = 'little'
```

Chapter 5

Optimization tips

On this chapter, you will get deeper knowledge of `PyTables` internals. `PyTables` has several places where the user can improve the performance of his application. If you are planning to deal with really large data, you should read carefully this section in order to learn how to get an important boost for your code. But if your dataset is small or medium size (say, up to 1 MB), you should not worry about that as the default parameters in `PyTables` are already tuned to handle that perfectly.

5.1 Taking advantage of Psyco

Psyco (see Rigo)is a kind of specialized compiler for Python that typically accelerates Python applications with no change in source code. You can think of Psyco as a kind of just-in-time (JIT) compiler, a little bit like Java's, that emit machine code on the fly instead of interpreting your Python program step by step. The result is that your unmodified Python programs run faster.

Psyco is very easy to install and use, so in most scenarios it is worth to have it a try. However, it only runs on Intel 386 architectures, so if you are using other architectures, you are out of luck (at least until Psyco will support yours).

As an example, imagine that you have a small script that reads and selects data over a series of datasets, like this:

```
def readFile(filename):
    "Select data from all the tables in filename"

    fileh = openFile(filename, mode = "r")
    result = []
    for table in fileh("/", 'Table'):
        result = [ p['var3'] for p in table if p['var2'] <= 20 ]

    fileh.close()
    return e

if __name__=="__main__":
    print readFile("myfile.h5")
```

In order to accelerate this piece of code, you can rewrite your main program to look like:

```
if __name__=="__main__":
    import pysco
    psyco.bind(readFile)
    print readFile("myfile.h5")
```

That's all!. From now on, each time that you execute your python script, Psyco will deploy its sophisticated algorithms so as to accelerate your calculations.

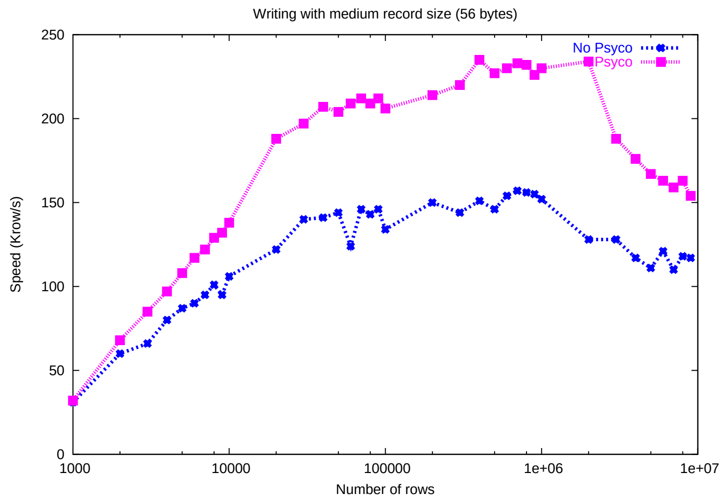

Figure 5.1: Writing tables with/without Psyco.

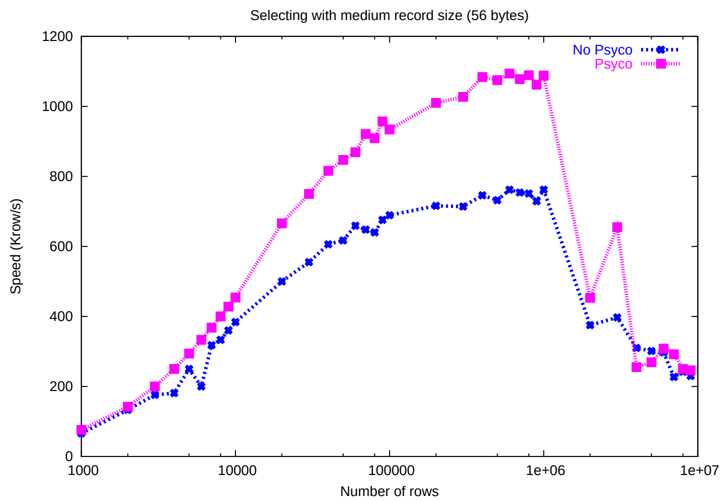

Figure 5.2: Reading tables with/without Psyco.

You can see in the graphs 5.1 and 5.2 how much I/O speed improvement you can get by using Psyco. By looking at this figures you can get an idea if these improvements are of your interest or not. In general, if you are not going to use compression you will take advantage of Psyco if your tables are medium sized (1e+3 < nrows < 1e+6), and this advantage will disappear progressively when the number of rows grows well over one million. However if you use compression, you will probably see improvements even beyond this limit (see section 5.2). As always, there is no substitute for experimentation with your own dataset.

5.2 Compression issues

One of the beauties of `PyTables` is that it supports compression on tables (but not on arrays!, that may come later), although it is disabled by default. Compression of big amounts of data might be a bit controversial feature, because compression has a legend of being a very big CPU time resources consumer. However, if you are willing to check if compression can help not only reducing your dataset file size but **also** improving your I/O efficiency, keep reading.

There is an usual scenario where users need to save duplicated data in some record fields, while the others

Table 5.1: Comparison between different compression libraries. The tests has been conducted on a Pentium 4 at 2 GHz and a hard disk at 4200 RPM.

Compr. Lib	File size (MB)	Time writing (s)	Time reading (s)	Speed writing (Krow/s)	Speed reading (Krow/s)
NO COMPR	244.0	24.4	16.0	18.0	27.8
Zlib (lvl 1)	8.5	17.0	3.11	26.5	144.4
Zlib (lvl 6)	7.1	20.1	3.10	22.4	144.9
Zlib (lvl 9)	7.2	42.5	3.10	10.6	145.1
LZO (lvl 1)	9.7	14.6	1.95	30.6	230.5
UCL (lvl 1)	6.9	38.3	2.58	11.7	185.4

have varying values. In a relational database approach such a redundant data can normally be moved to other tables and a relationship between the rows on the separate tables can be created. But that takes analysis and implementation time, and made the underlying libraries more complex and slower.

`PyTables` transparent compression allows the user to not worry about finding which is their optimum data tables strategy, but rather use less, not directly related, tables with a larger number of columns while still not cluttering the database too much with duplicated data (compression is responsible to avoid that). As a side effect, data selections can be made more easily because you have more fields available in a single table, and they can be referred in the same loop. This process may normally end in a simpler, yet powerful manner to process your data (although you should still be careful about what kind of scenarios compression use is convenient or not).

The compression library used by default is the **Zlib** (see Gailly and Adler), and as HDF5 *requires* it, you can safely use it and expect that your HDF5 files can be read on any other platform that has HDF5 libraries installed. Zlib provides good compression ratio, although somewhat slow, and reasonably fast decompression. Because of that, it is a good candidate to be used for compress you data.

However, in many situations (i.e. write *once*, read *multiple*), it is critical to have *very good* decompression speed (at expense of whether less compression or more CPU wasted on compression, as we will see soon). This is why support for two additional compressors has been added to PyTables: LZO and UCL (see Oberhumer). Following his author (and checked by the author of this manual), LZO offers pretty fast compression (although small compression ratio) and extremely fast decompression while UCL achieve an excellent compression ratio (at the price of spending much more CPU time) while allowing very fast decompression (and *very close* to the LZO one). In fact, LZO and UCL are so fast when decompressing that, in general (that depends on your data, of course), writing and reading a compressed table is actually faster (and sometimes **much faster**) than if it is uncompressed. This fact is very important, specially if you have to deal with very large amounts of data.

Be aware that the LZO and UCL support in PyTables is not standard on HDF5, so if you are going to use your PyTables files in other contexts different from PyTables you will not be able to read them.

In order to give you a raw idea of what ratios would be achieved, and what resources would be consumed, look at the table 5.1. This table has been obtained from synthetic data and with a somewhat outdated PyTables version (0.5), so take this just as a guide because your mileage will probably vary. Have also a look at the graphs 5.3 and 5.4 (these graphs has been obtained with tables with different row sizes and PyTables version than the previous example, so, do not try to directly compare the figures). They show how evolves the speed of writing/reading rows as the size (the row number) of tables grows. Even though in these graphs the size of one single row is 56 bytes, you can most probably extrapolate this figures to other row sizes. If you are curious how well can perform compression together with Psyco, look at the graphs 5.5 and 5.6. As you can see, the results are pretty interesting.

By looking at graphs, you can expect that, generally speaking, LZO would be the fastest both compressing and uncompressing, but the one that achieves the worse compression ratio (although that may be just ok for many situations). UCL is the slowest when compressing, but is faster than Zlib when decompressing, and, besides, it achieves very good compression ratios (generally better than Zlib). Zlib represents a balance between them: it's somewhat slow compressing, the slowest during decompressing, but it normally achieves fairly good compression ratios.

So, if your ultimate goal is reading as fast as possible, choose LZO. If you want to reduce as much as

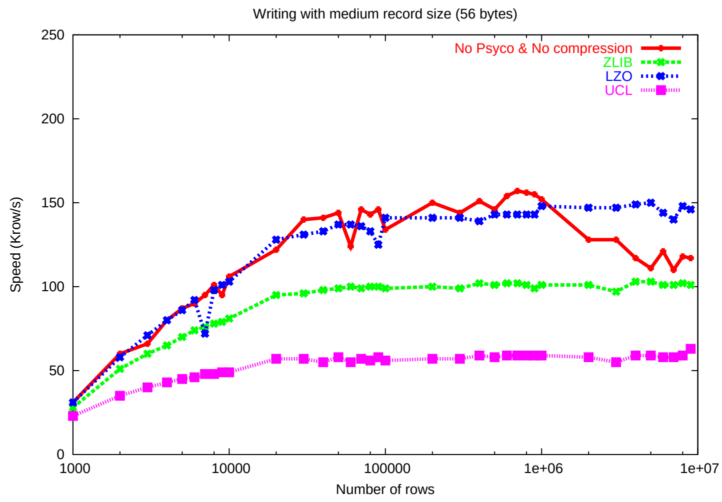

Figure 5.3: Writing tables with several compressors.

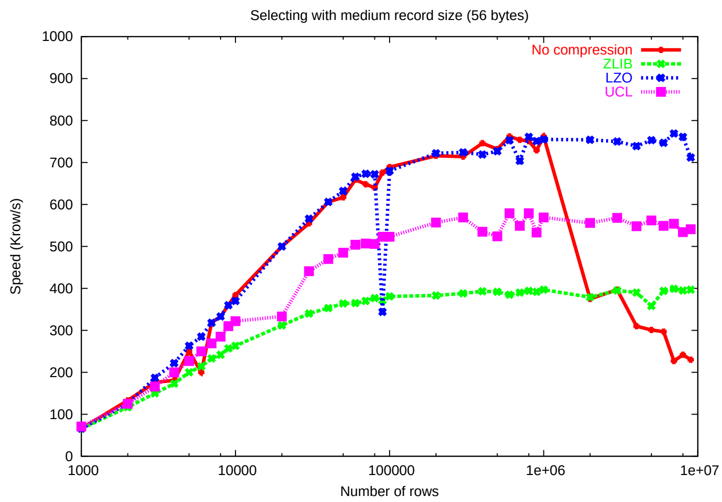

Figure 5.4: Reading tables with several compressors.

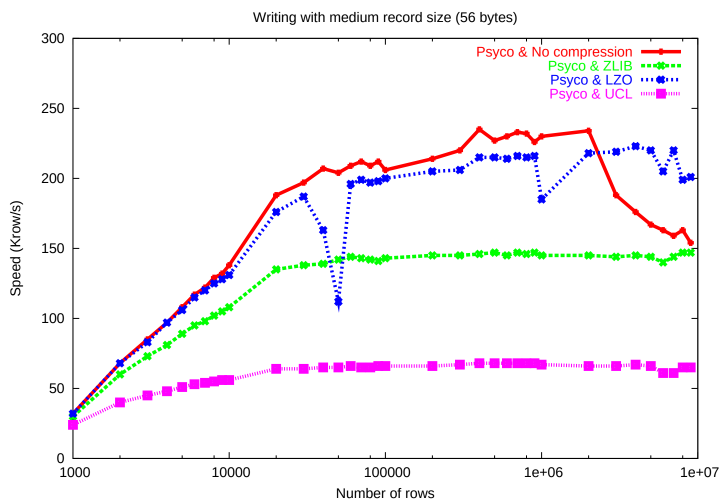

Figure 5.5: Writing tables with several compressors and Psyco.

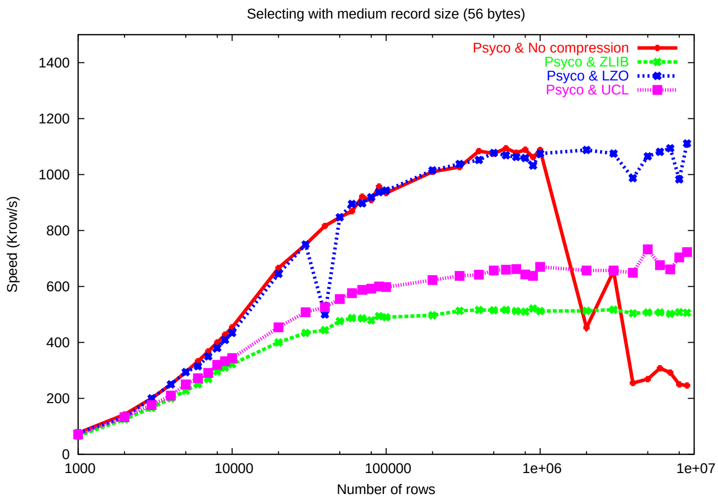

Figure 5.6: Reading tables with several compressors and Psyco.

possible your data, while retaining good read speed, choose UCL. If you don't mind too much about the above parameters and/or portability is important for you, Zlib is your best bet.

The compression level that I recommend to use for all compression libraries is 1. This is the lowest level of compression, but if you take the approach suggested above, normally the redundant data is to be found in the same row, so the redundant data locality is very high and such a small level of compression should be enough to achieve a good compression ratio on your data tables, saving CPU cycles for doing other things. Nonetheless, in some situations you may want to check how compression level affects your application.

You can select the compression library and level by setting the `complib` and `compress` keywords in the `Filters` class (see 4.12.1). A compression level of 0 will completely disable compression (the default), 1 is the less CPU time demanding level, while 9 is the maximum level and most CPU intensive. Finally, have in mind that LZO is not accepting a compression level right now, so, when using LZO, 0 means that compression is not active, and any other value means that LZO is active.

5.3 Shuffling (or how to make the compression even more effective)

The `HDF5` library provides an interesting filter that can leverage the results of your favorite compressor. Its name is *shuffle*, and because it can greatly benefit compression and don't take many CPU resources, it is active by *default* in `PyTables` whenever the compression is activated (independently of the compressor choosed). It is of course deactivated when compression is off (which is the default, as you already should know).

From the HDF5 reference manual:

> The *shuffle* filter de-interlaces a block of data by reordering the bytes. All the bytes from one consistent byte position of each data element are placed together in one block; all bytes from a second consistent byte position of each data element are placed together a second block; etc. For example, given three data elements of a 4-byte datatype stored as 012301230123, shuffling will re-order data as 000111222333. This can be a valuable step in an effective compression algorithm because the bytes in each byte position are often closely related to each other and putting them together can increase the compression ratio.

In table 5.2 you can see a benchmark that shows how the *shuffle* filter can help to the different libraries to compress data in three table datasets. Generally speaking, *shuffle* makes the writing process (shuffling+compressing) faster (between 7% and 22%), which is an interesting result in itself. However, the reading process (unshuffling+decompressing) is slower, but by a lesser extent (between 3% and 18%).

Table 5.2: Comparison between different compression libraries. The tests has been conducted on a Pentium 4 at 2 GHz and a hard disk at 4200 RPM.

Compr. Lib	File size (MB)	Time writing (s)	Time reading (s)	Speed writing (MB/s)	Speed reading (MB/s)
NO COMPR	165.4	24.5	17.13	6.6	9.6
Zlib (lvl 1)	26.4	22.2	5.77	7.3	28.4
Zlib+shuffle	4.0	19.0	5.94	8.6	27.6
LZO (lvl 1)	44.9	17.8	4.13	9.2	39.7
LZO+shuffle	4.3	16.4	5.03	9.9	32.6
UCL (lvl 1)	27.4	48.8	5.02	3.3	32.7
UCL+shuffle	3.5	38.1	5.31	4.3	30.9

But the most impressive is the level of compression that compressor filters can achieve after *shuffle* has passed over the data: the total file size can be up to 40 times smaller than the uncompressed file, and up to 5 times smaller than the already compressed files (!). Of course, the data for doing this test is synthetic, and *shuffle* seems to do a great work with it, so in general, the results will vary in your case. However, due to the small drawbacks (read are slowed down by a small extent) and its potential gains (faster writing, but specially much better compression level), I do believe that it is a good thing to have such a filter enabled by default in the battle for discovering redundancy in your data.

5.4 Informing `PyTables` about expected number of rows in tables

The underlying HDF5 library that is used by `PyTables` takes the data in bunches of a certain length, so-called *chunks*, to write them on disk as a whole, i.e. the HDF5 library treats chunks as atomic objects and disk I/O is always made in terms of complete chunks. This allows data filters to be defined by the application to perform tasks such as compression, encryption, checksumming, etc. on entire chunks.

An in-memory B-tree is used to map chunk structures on disk. The more chunks that are allocated for a dataset the larger the B-tree. Large B-trees take memory and causes file storage overhead as well as more disk I/O and higher contention for the metadata cache. Consequently, it's important to balance between memory and I/O overhead (small B-trees) and time to access to data (big B-trees).

`PyTables` can determine an optimum chunk size to make B-trees adequate to your dataset size if you help it by providing an estimation of the number of rows for a table. This must be made in table creation time by passing this value in the `expectedrows` keyword of `createTable` method (see 4.2.2).

When your table size is bigger than 1 MB (take this figure only as a reference, not strictly), by providing this guess of the number of rows you will be optimizing the access to your data. When the table size is larger than, say 100MB, you are **strongly** suggested to provide such a guess; failing to do that may cause your application doing very slow I/O operations and demanding **huge** amounts of memory. You have been warned!.

5.5 Selecting an User Entry Point (UEP) in your tree

If you have a **huge** tree in your data file with many nodes on it, creating the object tree would take long time. Many times, however, you are interested only in access to a part of the complete tree, so you won't strictly need PyTables to build the entire object tree in-memory, but only the *interesting* part.

This is where the `rootUEP` parameter of `openFile` function (see 4.1.2) can be helpful. Imagine that you have a file called `"test.h5"` with the associated tree that you can see in figure 5.7, and you are interested only in the section marked in red. You can avoid the build of all the object tree by saying to `openFile` that your root will be the `/Group2/Group3` group. That is:

```
fileh = openFile("test.h5", rootUEP="/Group2/Group3")
```

As a result, the actual object tree built will be like the one that can be seen in figure 5.8.

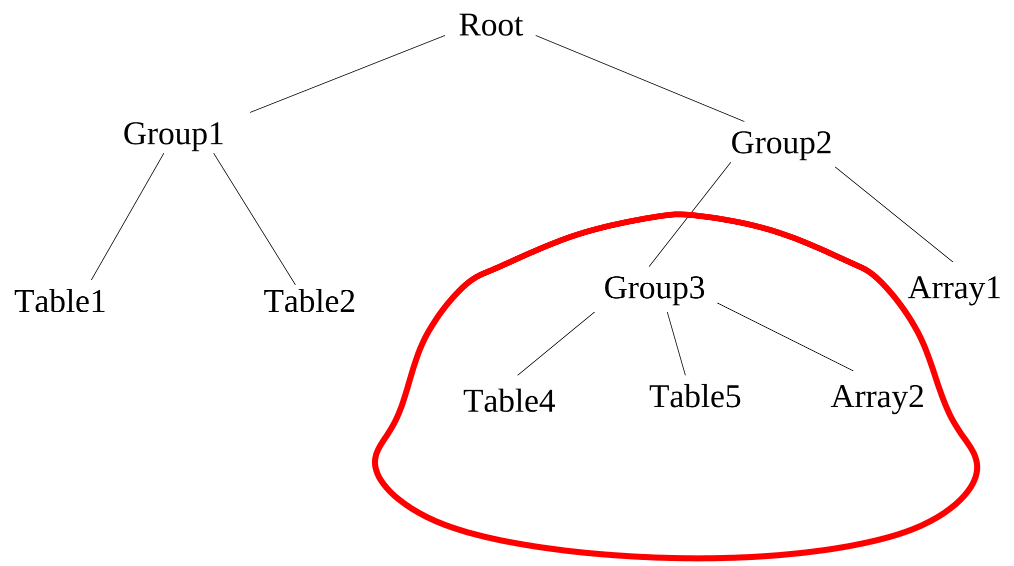

Figure 5.7: Complete tree in file `test.h5`, and subtree of interest for the user.

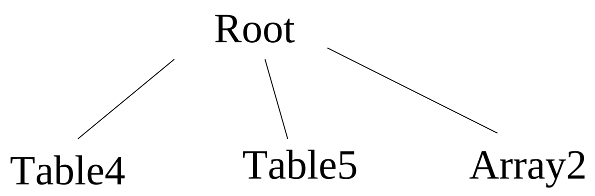

Figure 5.8: Resulting object tree derived from the use of the `rootUEP` parameter.

Of course this has been a simple example and the use of the `rootUEP` parameter was not very necessary. But when you have *thousands* of nodes on a tree, you will certainly appreciate the `rootUEP` parameter.

Appendix A

Supported data types in `PyTables`

`IsDescription` subclasses supports a limited set of data types to define the table fields. Such a set is roughly the same than the types supported by the `numarray` package (see Greenfield *et al.*) in Python, with the exception of the complex datatypes that are not supported yet.

These data types in table columns can be set through the use of the `Col` class and its descendants (see 4.11.2). You may find useful the table A as a quick reference to the complete set of supported data types in PyTables.

Table A.1: Data types supported by subclasses of `IsDescription` definitions.

Type Code	Description	C Type	Size (in bytes)	Python Counterpart
Bool	boolean	unsigned char	1	Boolean
Int8	8-bit integer	signed char	1	Integer
UInt8	8-bit unsigned integer	unsigned char	1	Integer
Int16	16-bit integer	short	2	Integer
UInt16	16-bit unsigned integer	unsigned short	2	Integer
Int32	integer	int	4	Integer
UInt32	unsigned integer	unsigned int	4	Long
Int64	64-bit integer	long long	8	Long
UInt64	unsigned 64-bit integer	unsigned long long	8	Long
Float32	single-precision float	float	4	Float
Float64	double-precision float	double	8	Float
CharType	arbitrary length string	char[]	*	String

Appendix B

PyTables File Format

PyTables has a powerful capability to deal with native HDF5 files created with another tools. However, there are situations were you may want to create truly native PyTables files with those tools while retaining fully compatibility with PyTables format. That is perfectly possible, and in this appendix is presented the format that you should endow to your own-generated files in order to get a fully PyTables compatible file.

We are going to describe the **1.2 version of PyTables file format** (introduced in PyTables version 0.8). At this stage, this file format is considered stable enough to do not introduce significant changes during a reasonable amount of time. As times goes by, some changes will be introduced (and documented here) in order to cope with new necessities. However, the changes will be carefully analyzed so as to ensure backward compatibility whenever is possible.

A PyTables file is composed with arbitrarily large amounts of HDF5 groups (Groups in PyTables naming scheme) and datasets (Leaves in PyTables naming scheme). For groups, the only requirements are that they must have some *system attributes* available. By convention, system attributes in PyTables are written in upper case, and user attributes in lower case but this is not enforced by the software. In the case of datasets, besides the mandatory system attributes, some conditions are further needed in their storage layout, as well as in the datatypes used in there, as we will see shortly.

As a final remark, you can use any filter as you want to create a PyTables file, provided that the filter is a standard one in HDF5, like *zlib*, *shuffle* or *szip* (although the last one cannot be used from within PyTables to create a new file, datasets compressed with szip can be read, because it is the HDF5 library which do the decompression transparently).

B.1 Mandatory attributes for a File

The File object is, in fact, an special HDF5 *group* structure that is *root* for the rest of the objects on the object tree. The next attributes are mandatory for the HDF5 *root group* structure in PyTables files:

CLASS This attribute should always be set to 'GROUP' for group structures.

PYTABLES_FORMAT_VERSION It represents the internal format version, and currently should be set to the '1.2' string.

TITLE A string where the user can put some description on what is this group used for.

VERSION Should contains the string '1.0'.

B.2 Mandatory attributes for a Group

The next attributes are mandatory for *group* structures:

CLASS This attribute should always be set to 'GROUP' for group structures.

TITLE A string where the user can put some description on what is this group used for.

VERSION Should contains the string '1.0'.

There exist a special Group, called the *root*, that, in addition to the attributes listed above, it requires the next one:

PYTABLES_FORMAT_VERSION It represents the internal format version, and currently should be set to the '1.2' string.

B.3 Mandatory attributes, storage layout and supported datatypes for Leaves

This depends on the kind of Leaf. The format for each type follows.

B.3.1 Table format

Mandatory attributes

The next attributes are mandatory for *table* structures:

CLASS Must be set to 'TABLE'.

TITLE A string where the user can put some description on what is this dataset used for.

VERSION Should contain the string '2.1'.

FIELD_X_NAME It contains the names of the different fields. The X means the number of the field (beware, order do matter). You should add as many attributes of this kind as fields you have in your records.

NROWS This should contain the number of *compound* datatype entries in the dataset. It must be an *int* datatype.

Storage Layout

A Table has a *dataspace* with a *1-dimensional chunked* layout.

Datatypes supported

The datatype of the elements (rows) of Table must be the H5T_COMPOUND *compound* datatype, and each of these compound components must be built with only the next HDF5 datatypes *classes*:

H5T_BITFIELD This class is used to represent the Bool type. Such a type must be build using a H5T_NATIVE_B8 datatype, followed by a HDF5 H5Tset_precision call to set its precision to be just 1 bit.

H5T_INTEGER This includes the next datatypes:

- **H5T_NATIVE_SCHAR** This represents a *signed char* C type, but it is effectively used to represent an Int8 type.
- **H5T_NATIVE_UCHAR** This represents an *unsigned char* C type, but it is effectively used to represent an UInt8 type.
- **H5T_NATIVE_SHORT** This represents a *short* C type, and it is effectively used to represent an Int16 type.
- **H5T_NATIVE_USHORT** This represents an *unsigned short* C type, and it is effectively used to represent an UInt16 type.
- **H5T_NATIVE_INT** This represents an *int* C type, and it is effectively used to represent an Int32 type.

- **H5T_NATIVE_UINT** This represents an *unsigned int* C type, and it is effectively used to represent an `UInt32` type.
- **H5T_NATIVE_LONG** This represents a *long* C type, and it is effectively used to represent an `Int32` or an `Int64`, depending on whether you are running a 32-bit or 64-bit architecture.
- **H5T_NATIVE_ULONG** This represents an *unsigned long* C type, and it is effectively used to represent an `UInt32` or an `UInt64`, depending on whether you are running a 32-bit or 64-bit architecture.
- **H5T_NATIVE_LLONG** This represents a *long long* C type (`__int64`, if you are using a Windows system) and it is effectively used to represent an `Int64` type.
- **H5T_NATIVE_ULLONG** This represents an *unsigned long long* C type (beware: this type does not have a correspondence on Windows systems) and it is effectively used to represent an `UInt64` type.

H5T_FLOAT This includes the next datatypes:

- **H5T_NATIVE_FLOAT** This represents a *float* C type and it is effectively used to represent an `Float32` type.
- **H5T_NATIVE_DOUBLE** This represents a *double* C type and it is effectively used to represent an `Float64` type.

H5T_STRING The datatype used to describe strings in PyTables is H5T_C_S1 (i.e. a *string* C type) followed with a call to the HDF5 `H5Tset_size()` function to set their length.

H5T_ARRAY This allows the construction of homogeneous, multi-dimensional arrays, so that you can include such objects in compound records. The types supported as elements of H5T_ARRAY datatypes are the ones described above. Currently, PyTables does not support nested H5T_ARRAY types.

You should note that *nested compound* datatypes are not allowed in `Table` objects.

B.3.2 `Array` format

Mandatory attributes

The next attributes are mandatory for *array* structures:

CLASS Must be set to `'ARRAY'`.

FLAVOR This is meant to provide the information about the kind of object kept in the `Array`, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take one the next string values:

- **"NumArray"** The dataset will be returned as a `NumArray` object (from the `numarray` package).
- **"CharArray"** The dataset will be returned as a `CharArray` object (from the `numarray` package).
- **"Numeric"** The dataset will be returned as an `array` object (from the `Numeric` package).
- **"List"** The dataset will be returned as a Python `List` object.
- **"Tuple"** The dataset will be returned as a Python `Tuple` object.
- **"Int"** The dataset will be returned as a Python `Int` object. This is meant mainly for scalar (i.e. without dimensions) integer values.
- **"Float"** The dataset will be returned as a Python `Float` object. This is meant mainly for scalar (i.e. without dimensions) floating point values.
- **"String"** The dataset will be returned as a Python `String` object. This is meant mainly for scalar (i.e. without dimensions) string values.

TITLE A string where the user can put some description on what is this dataset used for.

VERSION Should contain the string `'2.0'`.

Storage Layout

An `Array` has a *dataspace* with a *N-dimensional contiguous* layout (if you prefer a *chunked* layout see `EArray` below).

Datatypes supported

The elements of `Array` must have HDF5 *atomic* datatypes, and can currently be one of the next HDF5 datatypes *classes*: H5T_BITFIELD, H5T_INTEGER, H5T_FLOAT and H5T_STRING. See the `Table` format description in section B.3.1 for more info about these types.

You should note that H5T_ARRAY class datatypes are not allowed in `Array` objects.

B.3.3 `EArray` format

Mandatory attributes

The next attributes are mandatory for *earray* structures:

CLASS Must be set to `'EARRAY'`.

EXTDIM (*Integer*) Must be set to the extensible dimension. Only one extensible dimension is supported right now.

FLAVOR This is meant to provide the information about the kind of objects kept in the `EArray`, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take the same values as the `Array` object (see B.3.2),

except `"Int"` and `"Float"`.

TITLE A string where the user can put some description on what is this dataset used for.

VERSION Should contain the string `'1.0'`.

Storage Layout

An `EArray` has a *dataspace* with a *N-dimensional chunked* layout.

Datatypes supported

The elements of `EArray` must have HDF5 *atomic* datatypes, and can currently be one of the next HDF5 datatypes *classes*: H5T_BITFIELD, H5T_INTEGER, H5T_FLOAT and H5T_STRING. See the `Table` format description in section B.3.1 for more info about these types.

You should note that H5T_ARRAY class datatypes are not allowed in `EArray` objects.

B.3.4 `VLArray` format

Mandatory attributes

The next attributes are mandatory for *vlarray* structures:

CLASS Must be set to `'VLARRAY'`.

FLAVOR This is meant to provide the information about the kind of objects kept in the `VLArray`, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take one of the next values:

- **"NumArray"** The elements in dataset will be returned as `NumArray` objects (from the `numarray` package).
- **"CharArray"** The elements in dataset will be returned as `CharArray` objects (from the `numarray` package).

"String" The elements in the dataset will be returned as Python `String` objects of *fixed* length (and not as `CharArrays`).

"Numeric" The elements in the dataset will be returned as `array` objects (from the `Numeric` package).

"List" The elements in the dataset will be returned as Python `List` objects.

"Tuple" The elements in the dataset will be returned as Python `Tuple` objects.

"Object" The elements in the dataset will be interpreted as pickled (i.e. serialized objects through the use of the `Pickle` Python module) objects and returned as Python *generic* objects. Only one of such objects will be supported per entry. As the `Pickle` module is not normally available in other languages, this flavor won't be useful in general.

"VLString" The elements in the dataset will be returned as Python `String` objects of *any* length, with the twist that **Unicode** strings are supported as well (provided you use the **UTF-8** codification, see below). However, only one of such objects will be supported per entry.

TITLE A string where the user can put some description on what is this dataset used for.

VERSION Should contain the string `'1.0'`.

Storage Layout

An `VLArray` has a *dataspace* with a *1-dimensional chunked* layout.

Datatypes supported

The datatype of the elements (rows) of `VLArray` objects must be the H5T_VLEN *variable-length* (or VL for short) datatype, and the base datatype specified for the VL datatype can be of any *atomic* HDF5 datatype that is listed in the `Table` format description section B.3.1. That includes the classes:

- H5T_BITFIELD
- H5T_INTEGER
- H5T_FLOAT
- H5T_STRING
- H5T_ARRAY

You should note that this does not include another VL datatype, or compound datatype. Note as well that, for `Object` and `VLString` special flavors, the base for the VL datatype is always a H5T_NATIVE_UCHAR. That means that the complete row entry in the dataset has to be used in order to fully serialize the object or the variable length string.

In addition, if you plan to use a `VLString` flavor for your text data and you are using ascii-7 (7 bits ASCII) codification for your strings, but you don't know (or just don't want) to convert it to the required UTF-8 codification, you should not worry too much about that because the ASCII characters with values in the range [0x00, 0x7f] are directly mapped to Unicode characters in the range [U+0000, U+007F] and the UTF-8 encoding has the useful property that an UTF-8 encoded ascii-7 string is indistinguishable from a traditional ascii-7 string. So, you will not need any further conversion in order to save your ascii-7 strings and have an `VLString` flavor.

Bibliography

ASCHER, David, Paul F. DUBOIS, Konrad HINSEN, Jim HUGUNIN, and Travis OLIPHANT, : *Numerical Python*. Package to speed-up arithmetic operations on arrays of numbers.
URL `http://www.pfdubois.com/numpy/` 2, 7

EWING, Greg, : *Pyrex. A Language for Writing Python Extension Modules*.
URL `http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex` 7

GAILLY, JeanLoup and Mark ADLER, : *zlib. A Massively Spiffy Yet Delicately Unobtrusive Compression Library*. A standard library for compression purposes.
URL `http://www.gzip.org/zlib/` 7, 57

GREENFIELD, Perry, Todd MILLER, Richard L. WHITE, and J. C. HSU., : *Numarray*. Reimplementation of Numeric which adds the ability to efficiently manipulate large numeric arrays in ways similar to Matlab and IDL. Among others, Numarray provides the record array extension.
URL `http://stsdas.stsci.edu/numarray/` 2, 7, 44, 63

MERTZ, David, : *Objectify. On the 'Pythonic' treatment of XML documents as objects(II)*. Article describing XML Objectify, a Python module that allows working with XML documents as Python objects. Some of the ideas presented here are used in PyTables.
URL `http://www-106.ibm.com/developerworks/xml/library/xml-matters2/index.html` 3

NCSA, : *What is HDF5?* Concise description about HDF5 capabilities and its differences from earlier versions (HDF4).
URL `http://hdf.ncsa.uiuc.edu/whatishdf5.html` 1

OBERHUMER, Markus F.X.J., : *LZO and UCL. A couple of portable lossless data compression libraries*. They offer pretty fast compression and extremly fast decompression.
URL `http://www.oberhumer.com/opensource/` 7, 57

RIGO, Armin, : *Psyco. A Python specializing compiler.* Run existing Python software faster, with no change in your source.
URL `http://psyco.sourceforge.net` 55