Francesc Alted

PyTables User's Guide

Alted, Francesc:

`PyTables` User's Guide

Day of print: November, 13th

Contents

Chapter 1

Introduction

The goal of PyTables is to enable the end user to manipulate easily scientific data **tables** and *Numerical Python* objects in a hierarchical structure. The foundation of the underlying hierachical data organization is the excellent `HDF5` library (`http://hdf.ncsa.uiuc.edu/HDF5`). Right now, PyTables provides limited support of all the HDF5 functions, but I hope to add the more interesting ones (for PyTables needs) in the near future. Nonetheless, this package is not intended to serve as a complete wrapper for the entire HDF5 API.

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* seems to be quite 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 scientifc applications, for example) in an efficient manner that reduces demand on CPU time and I/O.

In order to emulate records (C structs in HDF5) in Python PyTables implements a special metaclass object with the capability to detect errors in field assignments as well as range overflows. PyTables also provides a powerful interface to process table data. Records in tables are also known, in the `HDF5` naming scheme, as *compound* data types.

For example, you can define arbitrary records in Python simply by declaring a class with the name field and types information, like in:

```
class Particle(IsRecord):
    name        = '16s'  # 16-character String
    idnumber    = 'Q'    # unsigned long long (i.e. 64-bit integer)
    TDCcount    = 'B'    # unsigned byte
    ADCcount    = 'H'    # unsigned short integer
    grid_i      = 'i'    # integer
    grid_j      = 'i'    # integer
    pressure    = 'f'    # float  (single-precision)
    energy      = 'd'    # double (double-precision)
```

then, you will normally instantiate it, fill it with your values, and save (arbitrary large) collections of them in a file for persistent storage. After that, this data can be retrieved and post-processed quite easily with `PyTables` or even with another `HDF5` application.

1.1 Features

`PyTables` has the next capabilities:

- *Support of table entities:* Allows working with large number of records that don't fit in memory.
- *Support of* `Numerical Python` *arrays:* Numeric arrays are a very useful complement of tables to keep homogeneus table slices (like selections of table columns).

- *Supports a hierarchical data model:* That way, you can structure very clearly all your data. Pytables builds up an object tree in memory that replicates the underlying file structure and the access to the file objects is made by walking throughout the `PyTables` object tree, and manipulating them.
- *Incremental I/O:* It supports adding records to already created tables. So you won't need to book large amounts of memory to fill the entire table and then save it to disk but you can do that incrementally, even between different Python sessions.
- *Allows field name, data type and range checking:* If `PyTables` does not report an error, you can be confident that your data is probably ok.
- *Support of files bigger than 2 GB:* The underlying HDF5 library already can do that (if your platform supports the C long long integer, or, on Windows, __int64), and PyTables automatically inherits this capability.
- *Data compression:* It supports data compression (through the use of the zlib library) out of the box. This become important when you have repetitive data patterns and don't want to loose your time searching for an optimized way to save them (i.e. it saves you data organization analysis time).
- *Big-Endian/Low-Endian safety:* `PyTables` has been carefully coded (as HDF5 itself) with little-endian/big-endian byte orderings issues in mind . So, in principle, you can write a file in a big-endian machine and read it in other little-endian without problems[1].

Finally, it should noted that `PyTables` is not intended to merely be a high level wrapper of selected HDF5 functionality (for this, have a look at HL-HDF5, the Swedish Meteorological and Hydrological Institute effort to provide another python interface to HDF5; see reference 5), but to provide a flexible tool to deal with (very) large amounts of data (i.e., typically bigger than available memory) in tables (heterogeneus data types) and arrays (homogeneus data types) organized in a hierarchical, persistent disk storage. PyTables take advantage of the powerful object orientation and introspection capabilities offered by Python to bring all this power to the user in a friendly manner.

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. This is achieved by *dynamically* creating an object tree imitating the HDF5 structure on disk. That way, the access to the HDF5 objects is made by walking throughout the `PyTables` object tree, and manipulating them.

A key aspect of PyTables is that for accessing to the different nodes on the object tree a **natural naming** schema is used, i.e. the attributes of the objects that represent HDF5 elements are the same as the names of the element's children[2].

For example, to refer to a table object located on directory */subgroup2* of a PyTables file, and called *table3*, you can use the construction `file.root.subgroup2.table3`, which is very Pythonic and comfortable in many use cases. See the chapter 3 for a more detailed explanation.

You should note that not all the data present on file is loaded in `PyTables` tree, but only the *metadata* (i.e. data that actually describes the structure of the actual data). The actual data is not read until you ask for it in a particular node, but by making use of the object tree (the metadata) you can get information on the objects on disk, for example, table names, title, name fields, data types in fields, number of records, or, in the case of arrays, shapes, typecode, and so on. You can traverse the tree with the supplied methods, and when you find the data you are interested in you can read it and process it. In some sense, you can think of PyTables as a tool that provide the same introspection capabilities of Python objects, but applied to the persistent storage of large amounts of data.

[1] Well, I didn't actually test that in real world, but if you do, please, tell me.
[2] I've taken this simple but powerful idea from the excellent `Objectify` module by David Mertz (see references 6 and 7)

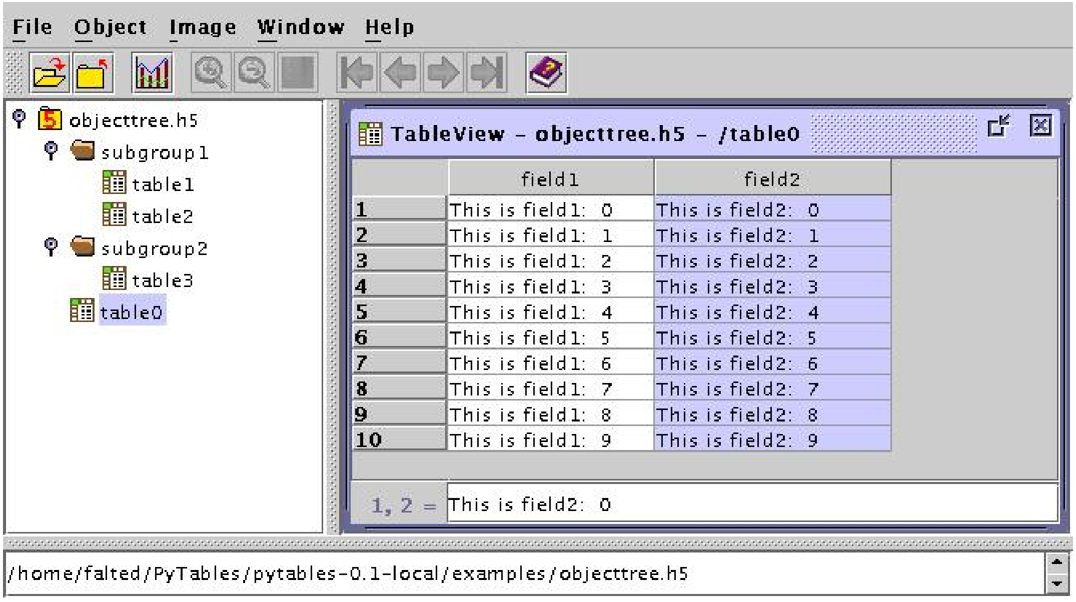

Figure 1.1: An HDF5 example with 2 subgroups and 3 tables.

To better understand the dynamic nature of this object tree, imagine we have made a script (in fact, this script exists; its name is `objecttree.py` and you can find it in the `examples/` directory) that creates a simple HDF5 file, with the structure that appears in figure 1.1 (we have used the `hdfview` application to obtain this image). During creation time, the object tree is updated (using only metadata, remeber) while data is being saved on file and when you close the file, this object is destroyed. If you re-open again this file (in read only mode, for example), the object tree with will be re-constructed from the metadata existent on file, and you can deal with it exactly in the same way than during the original creation process.

In figure 1.2 you can see an example of the object tree created by reading a PyTables file. If you are going to be a `PyTables` user, take your time to understand it[3]. That will also make you more proactive by avoiding programming mistakes.

[3] Bear in mind, however, that this diagram is **not** a standard UML class diagram; I've used an UML tool to draw it, that's all)

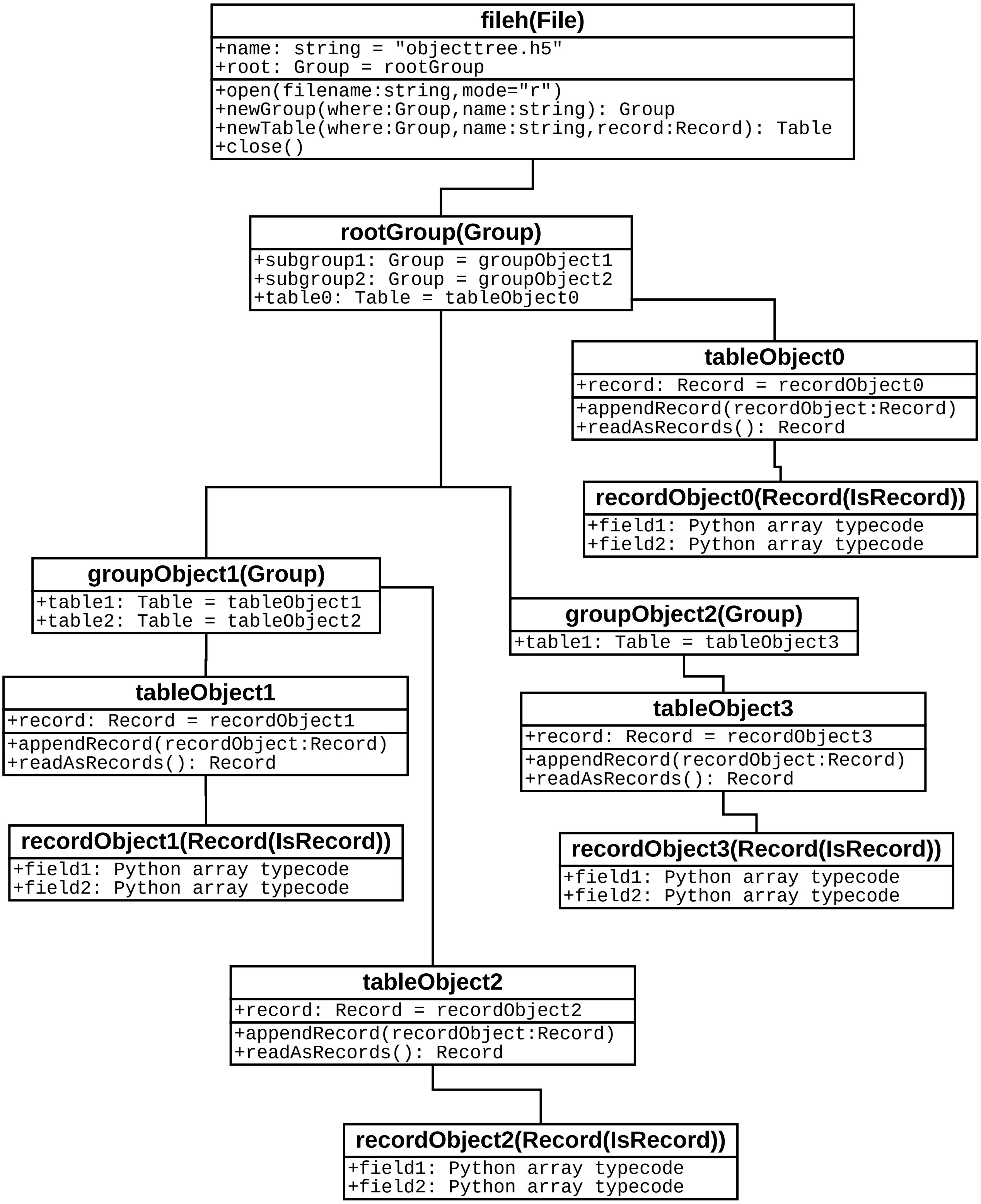

Figure 1.2: An object tree example in `PyTables`.

Chapter 2

Installation

This are instructions for Unix/Linux system. If you are using Windows, and you get the library to work, please tell me about.

Extensions in `PyTables` has been made using Pyrex (see reference 8) and C. You can rebuild everything from scratch if you got Pyrex installed, but this is not necessary, as the Pyrex compiled source is included in the distribution. In order to do that, merely replace `setup.py` script in these instructions by `setup-pyrex.py`.

The Python Distutils are used to build and install PyTables, so it is fairly simple to get things ready to go.

1. First, make sure that you have `HDF5 1.4.x` and `Numerical Python` installed (I'm using `HDF5 1.4.4` and `Numeric 22.0` currently). If don't, you can find them at `http://hdf.ncsa.uiuc.edu/HDF5` and `http://www.pfdubois.com/numpy`. Compile/install them.

   `setup.py` will detect `HDF5` libraries and include files under `/usr` or `/usr/local`; this will catch installations from RPMs, DEBs and most hand installations under Unix. If `setup.py` can't find your `libhdf5` or if you have several versions installed and want to select one of them, then you can give it a hint either in the environment (using the `HDF5_DIR` evironment variable) or on the command line by specifying the directory containing the include and lib directory. For example:

   ```
   --hdf5=/stuff/hdf5-1.4.4
   ```

   If your `HDF5` library was built as shared library, and if this shared library is 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/hdf5-1.4.4/lib"
   ```

   or perhaps just

   ```
   --lflags="-R /stuff/hdf5-1.4.4/lib"
   ```

   Check your compiler and linker documentation for correct syntax.

   It is also possible to specify linking against different libraries with the `--libs` switch:

   ```
   --libs="-lhdf5-1.4.6"
   --libs="-lhdf5-1.4.6 -lnsl"
   ```

2. From the main pytables distribution directory run this command, (plus any extra flags needed as discussed above):

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

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

3. To run the test suite change into the test directory and run this command, (assuming your shell is `sh` or compatible):

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

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

   If you would like to see some verbose output from the tests simply add the word `verbose` to the command line. You can also run only the tests in a particular test module by themselves. For example:

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

4. To install the entire `PyTables` Python package, change back to the root distribution directory and run this command as the root user:

   python setup.py install

That's it!. Now, read on the next section to see how to use PyTables.

Chapter 3

Usage

3.1 Getting started

This section is written in a tutorial style. We will see how to define our own records from Python and save collections of them (i.e. a **table**) on a file. Then, we will open this freshly created file and we will select some data in the table using Python cuts, creating Numerical arrays to keep this selection as separate objects in the tree. We will see how to browse the tree while retrieving metainformation about the actual data, and will finish by appending some rows to the existing table to show how table objects can be enlarged.

After reading this section you will hopefully learn the main features of PyTables. If you want more information on some specific instance variable, global function or method, go to the library reference in chapter 4. You can get deeper knowledge of PyTables by reading the other sections (XXX which ones?) in this chapter.

You will find in the directory `examples` the working version of all the code in this section (source file `tutorial.py`).

3.1.1 Importing `tables` objects

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

```
>>> import tables
>>>
```

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

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

which will export in your caller aplication namespace the next objects: `openFile`, `isHDF5`, `isPyTablesFile` and `IsRecord`. These are a rather small number of objects, and for commodity we will use this last way to access them.

If you are going to deal with `Numeric` arrays you will also need to import some objects from it. You can do that in the normal way. So, to access to PyTables functionality normally you should start you programs with:

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

3.1.2 Declaring a Record

Now, we want to declare a record object in order to save data that comes from our particle detector. Imagine that it 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. We will add two new fields called `grid_i` and `grid_j`. Our instrumentation also can obtain the pressure and energy of this particle, and we will add them in the same way. A simple-precision float will be enough to save `pressure` information, while `energy` would need a double-precision float. Finally, to track this particle we want to assign it a name to inform about the kind of the particle and a number identifier unique for each particle. So we will add a couple of fields: `name` will be the a string of up-to 16 characters and because we want to deal with a really huge number of particles, `idnumber` will be an integer of 64-bits.

With all of that, we declare a new Particle class that will keep all this info:

```
>>> class Particle(IsRecord):
...     name        = '16s'  # 16-character String
...     idnumber    = 'Q'    # unsigned long long (i.e. 64-bit integer)
...     TDCcount    = 'B'    # unsigned byte
...     ADCcount    = 'H'    # unsigned short integer
...     grid_i      = 'i'    # integer
...     grid_j      = 'i'    # integer
...     pressure    = 'f'    # float  (single-precision)
...     energy      = 'd'    # double (double-precision)
...
>>>
```

This definition class is quite auto-explanatory. Basically, you have to declare a class variable for each field you need, and as its value, the typecode for this data field. See appendix A for a list of typecodes supported in PyTables.

From now on, you can use this class instances as a container for your data, and as you will see shortly, you will get some magic properties associated with instances from `Particle`, and the class `IsRecord`[1] from which it is derived is responsible for that magic.

In order to do something useful with this record, we need to attach it to a `Table` object. But first, we must create a file where all the actual data pushed into `Table` will be saved.

3.1.3 Creating a PyTables file from scratch

To create a PyTables file use the first-level `openFile` funtion:

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

This `openFile` is one of the objects imported by the "`from tables import *`", do you remember?. We are telling that we want to create a new file called "`tutorial.h5`" in "`w`"rite mode and with an informative title string ("`Test file`"). This function tries to open this file, and if successful, returns a `File` instance which hosts the root of the object tree on its `root` attribute.

3.1.4 Creating a new group

Now, to better organize our data, we will create a group hanging from the root called *detector*. We will use this group to save our particle data there.

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

[1] `IsRecord` is actually a *metaclass* in object slang, but we don't need to explain nothing more about it now. Check the sources if you are interested on how that works.

Here, we have take the `File` instance `h5file` and invoked its `createGroup` method, telling that we want to create a new group called *detector* hanging from "/", which is other way to refer to the `h5file.root` object we mentioned before. This will create a new `Group` instance that will be assigned to the `group` variable.

3.1.5 Creating a new table

Let's now create the `Table` object hanging from the new created group. We do that by calling the `createTable` method from the `h5file` object:

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

You can see how we asked to create the `Table` instance hanging from `group`, with name *'readout'*. As the record object we have passed an instance of Particle, the class that we have declared before, and finally we attach it a "*Readout example*" title. With all this information, a new `Table` instance is created and assigned to `table` variable.

Now, time to fill this table with some values. But first, we want to get a pointer to the record object in this `table` instance:

```
>>> particle = table.record
```

The `record` attribute of `table` points to the `Particle` instance used to create the table, and we assign it to the `particle` variable that will be used as a shortcut. This step is not really necessary, but helps to code legibility (and allows me to introduce the `record` attribute).

We can proceed right now to the filling process:

```
>>> for i in xrange(10):
...     # First, assign the values to the Particle record
...     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)  # This exceeds long integer range
...     # Insert a new particle record
...     table.appendAsRecord(particle)
...
>>>
```

This code is quite easy to understand. The lines inside the loop just assigned values to the `particle` record object and then a call to the `appendAsRecord` method of `table` instance is made to put this information in the `table` I/O buffer.

After we have filled all our data, we must flush the I/O buffer for the table if we want to consolidate all this data on disk. We do that by calling the `table flush` method.

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

And last, we close the h5file File instance to close the file:

```
>>> h5file.close()
```

With all that, you have created your first PyTables file with a table inside it. That was easy, admit it. Now, you can have a look at it with some generic HDF5 tool, like h5dump or h5ls. Here is the result of passing to h5ls the `tutorial.h5` file:

```
$ h5ls -rd tutorial.h5
/tutorial.h5/detector    Group
/tutorial.h5/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}
```

or, using the "dumpFile.py" PyTables utility (located in examples/ directory):

```
$ python2.2 dumpFile.py tutorial.h5
Filename: tutorial.h5
All objects:
Filename: tutorial.h5 \\ Title: "Test file" \\ Format version: 1.0
/ (Group) "Test file"
/detector (Group) "Detector information"
/detector/readout Table(8, 10) "Readout example"
```

You can pass the `-v` option to `dumpFile.py` if you want more verbosity.

3.1.6 Reading and selecting table data

Ok. We have our data on disk. But we want to access it and select some values we are interested it in some specific columns. That's is easy to do. First, let's open the file we have recently created:

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

This time, we have opened the file in "a"ppend mode. We are using this mode because we want to add more information to the file. But for the moment, our interest is to read an select data from table on disk. The next lines do exactly that:

```
>>> table = h5file.root.detector.readout
>>> pressure = [ x.pressure for x in table.readAsRecords()
...                     if x.TDCcount > 3 and x.pressure < 50 ]
```

The first line is only to define a shortcut to the *readout* table which is a bit deeper on the object tree. As you can see, we have used the **natural naming** schema to access it. We could also have used the `h5file.getNode` method instead, and we certainly do that later on.

The last two lines are a Python comprenhensive list. It loops on records returned by `table.readAsRecords()` iterator that returns values until table data is exhausted. This records are filtered using the *cut* expression `x.TDCcount > 3 and x.pressure < 50`, and the `pressure` field for satisfying records is selected to form the final list that is assigned to `pressure` variable.

We could have used a normal `for` loop to do that, but I find comprehensions syntax more compact and elegant.

Let's select the names for the same set of particles:

```
>>> names = [ x.name for x in table.readAsRecords()
...                  if x.TDCcount > 3 and x.pressure < 50 ]
```

Ok. that's enough for selections. Now, save these selections on file.

3.1.7 Creating new array objects

In order to separate the selections from the detector data, we will create a new group, called `columns` hanging from the root group:

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

Note that this time we have specified the first parameter in a natural naming fashion (`h5file.root`) instead of a path string ("/").

Now, create the Array objects on file:

```
>>> h5file.createArray(gcolumns, ’pressure’, array(pressure),
...                     "Pressure column selection")
<tables.Array.Array object at 0x8217cac>
>>> h5file.createArray(’/columns’, ’name’, array(names),
...                     "Name column selection")
<tables.Array.Array object at 0x814c3dc>
```

We already know the first two parameters of the `createArray` methods (are the same as in `createTable`): they are the parent group where `Array` will be created and the `Array` instance name. You can figure out that the fourth parameter is the title. And in the third position we have the `Numeric` objects we want to save on disk. They are built from the selection lists we created before, and their typecodes are automatically selected by the `array()` constructor to store the list of values. In these case, they will become double-precision arrays, as we will see in short.

Note that createArray method returns an Array instance, that we don’t keep anywhere. Don’t worry, it is attached to the object tree, and can be easily retrieved later on. In fact, you can browse the object tree very easily and retrieve any data you may be interested in. Keep reading.

3.1.8 Traversing the object tree

PyTables, following the Python tradition, offers powerful instropection capabilities, i.e. you can easily ask information about any component of the object tree as well as traverse the tree searching for something. To start with, you can get a first glance image of the object tree, by simply printing the existing `File` instance:

```
>>> print h5file
Filename: tutorial.h5 \\ Title: "Test file" \\ Format version: 1.0
/ (Group) "Test file"
/columns (Group) "Pressure and Name"
/columns/name Array(4, 16) "Name column selection"
/columns/pressure Array(4,) "Pressure column selection"
/detector (Group) "Detector information"
/detector/readout Table(8, 10) "Readout example"
>>>
```

Right, it seems that all our objects are there. We can use the `walkGroups` method of `File` class to list all the groups in 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. And combining it with the `listNodes` method, we can do very powerful lists. Let’s see an example listing 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, 16) "Name column selection"
/columns/pressure Array(4,) "Pressure column selection"
```

`listNodes` lists all the nodes hanging from a `group`, and if *classname* keyword is specified, the method will filter all instances which are not representants of it. We have specified so as to return only the `Array` instances.

Caveat emptor: `listNodes` (conversely to `walkGroups`) returns an actual list, and not an iterator!.

As a final example, we will list all the `Leaf` (i.e. `Table` or `Array`) objects in */detector* group. Check that we have only one representant of `Table` class in this group:

```
>>> for table in h5file.listNodes("/detector", 'Leaf'):
...     print table
...
/detector/readout Table(8, 10) "Readout example"
>>>
```

Of course you can do more sophisticated node selections using this two powerful funtions, but first, we need to learn a bit about important instance variable of PyTables objects.

3.1.9 Getting object metadata

Each object in PyTables has metadata about the actual data on the file. Normally this metainformation is accessible through the node instance variables. Let's see some examples:

```
>>> # Get the "/detector/table"
... table = h5file.getNode("/detector/readout", classname = 'Table')
>>> # Get metadata from table
... print "Object:", table
Object: /detector/readout Table(8, 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: %d" % (table.nrows)
Number of rows in table: 10
>>> print "Table variable names (sorted alphanumerically) with their type:"
Table variable names (sorted alphanumerically) with their type:
>>> for i in range(len(table.varnames)):
...     print "  ", table.varnames[i], ':=', table.vartypes[i]
...
   ADCcount := H
   TDCcount := B
   energy := d
   grid_i := i
   grid_j := i
   idnumber := Q
   name := 16s
   pressure := f
```

Here, the `name`, `title`, `nrows`, `varnames` and `vartypes` attributes of `table` object give us quite a lot of information about actual table data.

Observe how we have used the `getNode` method of `File` class to access a node in the tree, instead of using the natural naming method. Both are useful, and depending on the context you will prefer to use one or another. `getNode` has the advantage that can get a node from the pathname string (like in this example), and you can force that the node in that location has to be a *classname* instance. However, natural naming is more elegant and quicker to specify (specially if you are using the name completion capability present in interactive console).

Now, print some metadata in */columns/pressure* Array object:

```
>>> # Get the object in "/columns pressure"
... pressureObject = h5file.getNode("/columns", "pressure")
>>>
>>> # Get some metadata on this object
... print "Info on the object:", pressureObject
Info on the object: /columns/pressure Array(4,) "Pressure column selection"
>>> print "  shape: ==>", pressureObject.shape
  shape: ==> (4,)
>>> print "  title: ==>", pressureObject.title
  title: ==> Pressure column selection
>>> print "  typecode ==>", pressureObject.typecode
  typecode ==> d
```

If you look at the `typecode` attribute of the `pressureObject`, you can certify that this is a "d"ouble `Numeric` array, and that by looking at their `shape` attribute the array on disk is unidimensional and has 4 elements.

3.1.10 Reading actual data from `Array` objects

Once you have found the desired `Array` and decided that you want to retrieve the actual `Numeric` array from it, you should use the `read` method of the `Array` object:

```
>>> # Read the 'pressure' actual data
... pressureArray = pressureObject.read()
>>>
>>> # Read the 'name' Array actual data
... nameArray = h5file.root.columns.name.read()
>>>
>>> # Check what kind of object we have created (they should be Numeric arrays)
... print "pressureArray is object of type:", type(pressureArray)
pressureArray is object of type: <type 'array'>
>>> print "nameArray is object of type:", type(nameArray)
nameArray is object of type: <type 'array'>
>>>
```

You can verify that `read()` returns an authentic `Numeric` array looking at the output of the `type()` call.

3.1.11 Appending data to an existing table

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

```
>>> # Create a shortcut to table object
... table = h5file.root.detector.readout
>>>
>>> # Get the object record from table
```

```
... particle = table.record
>>>
>>> # Append 5 new particles to table (yes, tables can be enlarged!)
... 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)  # This exceeds long integer range
...     table.appendAsRecord(particle)
...
>>> # Flush this table
... table.flush()
```

That works exactly in the same way than filling 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[2].

If you look carefully at the code you will see that we have used the `table.record` attribute to access to a `Particle` instance and that way we could use it to fill new values. However, it should be stressed that it is not necessary to have the original class definition (Particle) in our code to re-create it: it will be created only from metadata existing on file, and it behaves exactly as an original Particle instance!. This is part of the magic that allow the use of *metaclasses* in PyTables, and that will easy the creation of portable aplications that can read any PyTables file **regardless** of having access to the original Python record class definition.

Let's have a look at some columns of the resulting table:

```
>>> for x in table.readAsRecords():
...     print "%-16s | %11.1f | %11.4g | %6d | %6d | %8d |" % \
...        (x.name, x.pressure, x.energy, x.grid_i, x.grid_j,
...         x.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 |
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 |
>>> print
>>> print "Total numbers of entries after appending new rows:", table.nrows
Total numbers of entries after appending new rows: 15
```

In figure 3.1 you can see a view of the PyTables file we have created on this tutorial.

[2] Note that you can only append values to tables, not array objects. However, I plan to support unlimited dimension arrays in short term. Keep tuned.

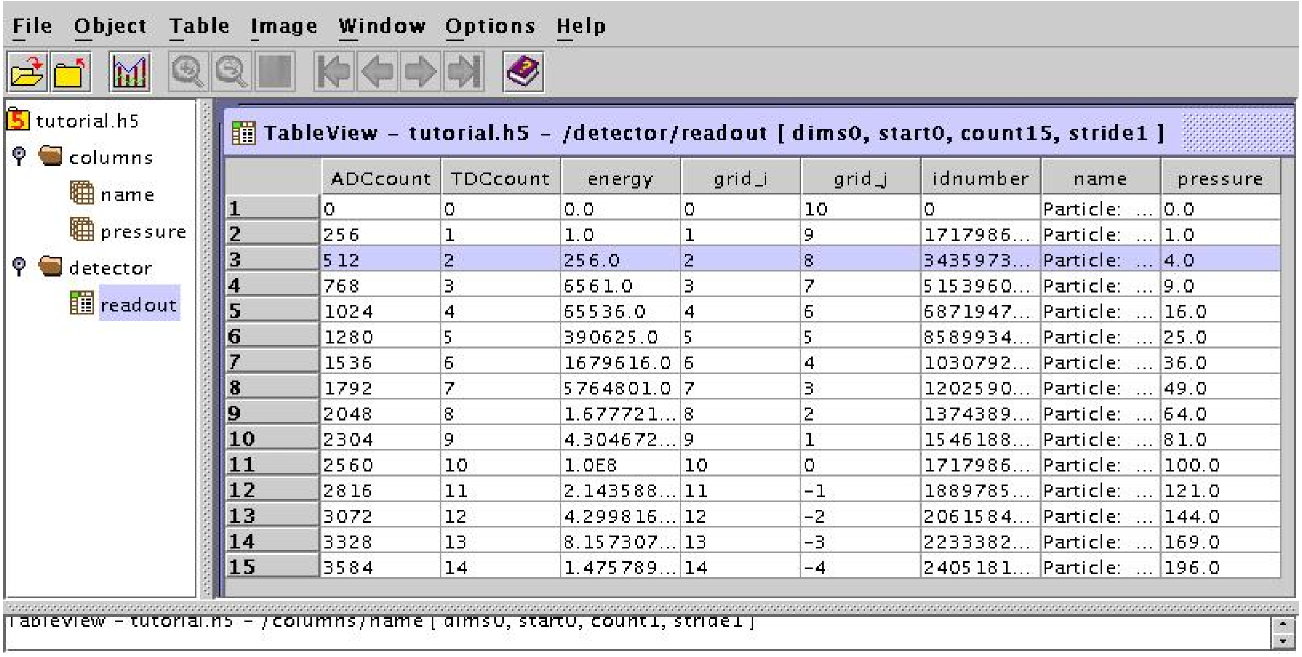

Figure 3.1: The PyTables file created in tutorial.

We are near the end of the tutorial. Ei!, do not forget to close the file after you finish all the work:

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

3.2 Checking for field name, data type and data range

Now, time for a more sophisticated example. Here, we will create a couple of directories (groups, in HDF5 jargon) hanging directly from the root directory called `Particles` and `Events`. Then, we will put 3 tables in each group; in `Particles` we will put instances of `Particle` records and in `Events`, instances of `Event`. After that, we will feed the tables with 257 (you will see soon why I choose such an "esoteric" number) entries each. Finally, we will read the recently created table `/Events/TEvent3` and select some values from it using a comprehension list.

See at the next script. It seems to do all of that, but a couple of small suprises will appear:

```
from tables import *
class Particle(IsRecord):
    name        = '16s'  # 16-character String
    lati        = 'i'    # integer
    longi       = 'i'    # integer
    pressure    = 'f'    # float  (single-precision)
    temperature = 'd'    # double (double-precision)
class Event(IsRecord):
    name        = '16s'  # 16-character String
    TDCcount    = 'B'    # unsigned char
    ADCcount    = 'H'    # unsigned short
    xcoord      = 'f'    # float  (single-precision)
    ycoord      = 'f'    # float  (single-precision)
```

```
# Open a file in "w"rite mode
fileh = openFile("example2.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 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.record
    # Fill the table with 10 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
        particle.pressure = float(i*i)
        particle.temperature = float(i**2)
        # This injects the Record values
        table.appendAsRecord(particle)
    # Flush the table buffers
    table.flush()
# Now, go for Events:
for tablename in ("TEvent1", "TEvent2", "TEvent3"):
    # Create a table in Events group
    table = fileh.createTable(root.Events, tablename, Event(),
                           "Events: "+tablename)
    # Get the record object associated with the table:
    event = table.record
    # Fill the table with 257 events
    for i in xrange(257):
        # First, assign the values to the Event record
        event.name  = 'Event: %6d' % (i)
        event.TDCcount = i
        event.ADCcount = i * 2
        event.xcoor = float(i**2)
        event.ycoord = float(i**4)
        # This injects the Record values
        table.appendAsRecord(event)
    # 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.readAsRecords()
      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()
```

If you have read the code carefully it looks pretty good, but it won't work. If you run this example, you will get the next error:

```
Traceback (most recent call last):
  File "example2.py", line 68, in ?
    event.xcoor  = float(i**2)
AttributeError: 'Event' object has no attribute 'xcoor'
```

This error is saying us that we tried to assign a value to a non-existent field in an `Event` object. By looking carefully at the `Event` attributes, we see that we misspelled the `xcoord` field (we wrote `xcoor` instead). After correcting this in the source, and running again, we find another problem:

```
Traceback (most recent call last):
  File "example2.py", line 69, in ?
    table.appendRecord(event)
  File "/usr/lib/python2.2/site-packages/tables/Table.py", line 210, in appendRecord
    self._v_packedtuples.append(recordObject._f_pack2())
  File "/usr/lib/python2.2/site-packages/tables/IsRecord.py", line 121, in _f_pack2
    self._f_raiseValueError()
  File "/usr/lib/python2.2/site-packages/tables/IsRecord.py", line 130, in
_f_raiseValueError
    raise ValueError, \
ValueError: Error packing record object:
 [('ADCcount', 'H', 256), ('TDCcount', 'B', 256), ('name', '16s', 'Event:    256'),
 ('xcoord', 'f', 65536.0), ('ycoord', 'f', 4294967296.0)]
 Error was: ubyte format requires 0<=number<=255
```

This other error is saying that one of the records is having trouble to be converted to the data types stated in the Event class definition. By looking carefully to the record object causing the problem, we see that we are trying to assign a value of 256 to the 'TDCcount' field which has a 'B' (C unsigned char) typecode and the allowed range for it is `0<=TDCcount<=255`. This is a very powerful capability to automatically check for ranges: the message error is explicit enough to figure out what is happening. In this case you can solve the problem by promoting the `TDCcount` to 'H' which is a unsigned 16-bit integer, or avoid the mistake you probably made in assigning a value greater than 255 to a 'B' typecode.

If we change the line:

```
event.TDCcount = i
```

by the next one:

```
event.TDCcount = i % (1<<8)
```

you will see that our problem has disappeared, and the HDF5 file has been created.

Finally, to test the type cheking, we will change the next line:

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

to read:

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

After this modification, the next exception will be raised when the script is executed:

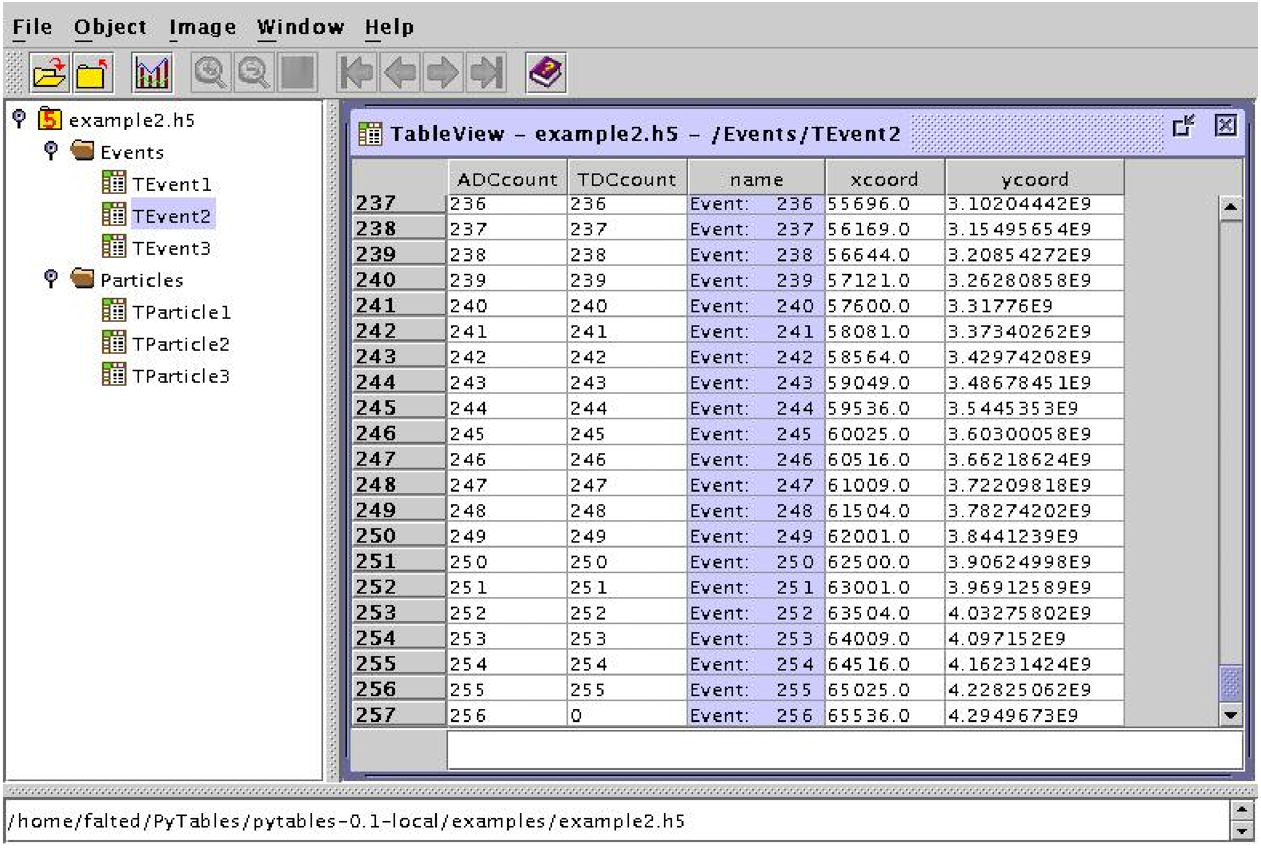

Figure 3.2: Table hierarchy for second example.

```
Traceback (most recent call last):
  File "tutorial2.py", line 68, in ?
    table.appendAsRecord(event)
  File "/home/falted/PyTables/pytables-0.2/tables/Table.py", line 279, in appendAsRecord
    self._v_packedtuples.append(RecordObject._f_pack2())
  File "/home/falted/PyTables/pytables-0.2/tables/IsRecord.py", line 181, in _f_pack2
    self._f_raiseValueError()
  File "/home/falted/PyTables/pytables-0.2/tables/IsRecord.py", line 135, in _f_raiseValue
    raise ValueError, \
ValueError: Error packing record object:
 [('ADCcount', 'H', '0'), ('TDCcount', 'B', 0), ('name', '16s', 'Event:      0'), ('xcoord
 Error was: required argument is not an integer
```

that states the error.

After correcting all this errors, admire the structure we have created in figure 3.2. As before, you will find in the directory `examples` the working version of the code (source file `tutorial2.py`).

Feel free to visit the rest of examples in directory `examples`, and try to understand them. I've tried to make several cases in different to give you an idea of the `PyTables` capabilities and its way of dealing with HDF5 objects.

Chapter 4

Library Reference

`PyTables` implements several classes to represent the different nodes in the object tree. They are called `File`, `Group`, `Leaf`, `Table` and `Array`. Another one is responsible to build record objects from a subclass user declaration, and performs field, type and range checks; it is called `IsRecord`. An important function, called `openFile` is responsible to create, open or append to `PyTables` files. In addition, a few utility funtions are defined to guess if an user supplied file is a `PyTables` file or not. These are called `isPyTablesFile` and `isHDF5`. Finally, several variables are also available to the user that informs about `PyTables` version, file format version or underlying libraries (as for example `HDF5`) version number.

Let's start discussing the global variables and functions available to the user, then the methods in the classes defined in PyTables.

4.1 `tables` Variables and Functions

4.1.1 Global Variables

__version__ The PyTables version number.

HDF5Version The underlying HDF5 library version number.

ExtVersion The Pyrex extension types version. This may be useful for reporting bugs.

4.1.2 Global Functions

openFile(filename, mode='r', title='') Open a PyTables file an returns a File object.

- **filename:** The name of the file (supports environment variable expansion). It must have any of `".h5"`, `".hdf"` or `".hdf5"` extensions.
- **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 is 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.

isHDF5(filename) Determines whether filename is in the HDF5 format. 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.

4.2 The `IsRecord` class

This class is in fact a so-called *metaclass* object. There is nothing special on it, except that their subclasses attributes are transformed during its construction phase, and new methods for the are defined based on the values of the attributes. In that way, we can *force* the resulting instance to only accept assignments on the declared attributes (in fact, it has a few more, but they are hidden with prefixes like `"__"`, `"_v_"` or `"_f_"`, so please, don't use attributes names starting with these prefixes). If you try to do an assignment to a non-declared attribute, `PyTables` will raise an error.

To use such a particular class, you have to declare a descendent class from *IsRecord*, with many attributes as fields you want in your record. To declare their types, you simply assign to these attributes their *typecode*. That's all, from now on, you can instantiate objects from you new class and use them as a very flexible record object with safe features like automatic name field, data type and range checks (see the section 3.2 for an example on how it works).

See the appendix A for a relation of data types supported in a `IsRecord` class declaration.

4.3 The `File` class

This class is returned when a PyTables is opened with the `openFile` function. It is in charge of create, open, flush and close the PyTables files. Also, File class offer methods to traverse the object tree, as well as to create new nodes. One of its attributes (`root`) represents the entry point to the object tree.

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

4.3.1 `File` instance variables

filename Filename opened.

mode Mode in which the filename was opened.

title The title of the root group in file.

root The root group in file. This is the entry point to the object tree.

4.3.2 `File` methods

createGroup(where, name, title='') Create a new Group instance with name *name* in *where* location.

- **where** The parent group where the new group will hang. *where* parameter can be a path string (for example `"/Particles/TParticle1"`), or another Group instance.
- **name** The name of the new group.
- **title** A descrition for this group.

createTable(where, name, RecordObject, title='', compress=3, expectedrows=10000) Create a new Table instance with name *name* in *where* location.

- **where** The parent group where the new table will hang. *where* parameter can be a path string (for example `"/Particles/TParticle1"`), or Group instance.

[1] On the following, the term `Leaf` will refer to a `Table` instance. Right now, the only supported `Leaf` objects are `Table` and `Array`, but this list may be increased in the future.

name The name of the new table.

RecordObject An instance of a user-defined class (derived from the `IsRecord` class) where table fields are defined.

title A descrition for this table.

compress Specifies a compress level for data. The allowed range is 0-9. A value of 0 disables compression. The default is compression level 3, that balances between compression effort and CPU consumption.

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

createArray(where, name, NumericObject, title='') Create a new instance Array with name *name* in *where* location.

where The parent group where the new array will hang. *where* parameter can be a path string (for example `"/Particles/TParticle1"`), or Group instance.

name The name of the new array.

NumericObject The Numeric array to be saved.

title A description for this table.

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 not 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. Allowed names in *classname* are: 'Group', 'Leaf', 'Table' and 'Array'.

listNodes(where, classname='') Returns a list with all the object nodes (Group or Leaf) hanging from *where*. The list is alphanumerically 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'` and `'Array'`.

walkGroups(where='/') *Iterator* that recursively obtains Groups (not Leaves) hanging from *where*. If *where* is not supplied, the root object is taken as origin. The groups are returned from top to bottom, and they are alphanumerically sorted when they are at the same level.

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

flush() Flush all the objects on all the HDF5 objects tree.

close() Flush all the objects in HDF5 file and close the file.

4.4 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 string (like in `'/group1/group2'`) or as a complete object path written in the Pythonic fashion known as *natural name* schema (like in `file.root.group1.group2`) and 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 to a Group object to no collide with existing children node names. For this reason and to not pollute the children namespace, it is explicitely forbidden to assign "normal" attributes to Group instances, and the only ones allowed must start with "`_c_`" (for class variables), "`_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.

4.4.1 `Group` class variables

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

_c_objgroups Dictionary with all object groups on tree.

_c_objleaves Dictionary with all object leaves on tree.

4.4.2 `Group` instance variables

_v_title A description for this group.

_v_name The name of this group.

_v_pathname A string representation of the group location in tree.

_v_parent The parent Group instance.

_v_objchilds Dictionary with all objects (groups or leaves) hanging from this instance.

_v_objgroups Dictionary with all object groups hanging from this instance.

_v_objleaves Dictionaly with all object leaves hanging from this instance.

4.4.3 `Group` methods

This 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_listNodes(classname='') Return a list with all the object nodes hanging from this instance. The list is alphanumerically 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'`.

_f_walkGroups() *Iterator* that recursively obtains Groups (not Leaves) hanging from self. The groups are returned from top to bottom, and are alphanumerically sorted when they are at the same level.

4.5 The `Leaf` class

This is a helper class useful to place common functionality of all Leaf objects. 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. Right now this set is composed by `Table` and `Array` objects. In fact, `Table` and `Array` classes inherit functionality from this class using the *mix-in* technique.

Normally the user will not need to call any method from here, but it is useful to know that it exists because it can be used as a filter in methods like `File.GetNode()` or `File.listNodes()`, against others.

4.6 The `Table` class

Instances of this class represents table objects in the object tree. It provides methods to create new tables or open existing ones, as well as methods to write/read data and metadata to/from table objects in the file.

Data can be written or read both as records or as tuples. Records are recommended because they are more intuitive and less error prone although they are slow. Using tuples (or value sequences) is faster, but the user must be very careful because when passing the sequence of values, they have to be in the correct order (alphanumerically ordered by field names). If not, unexpected results can appear (most probably `ValueError` exceptions will be raised).

4.6.1 `Table` instance variables

name The node name.

title The title for this node.

record The record object for this table.

nrows The number of rows (records) in this table.

varnames The field names for the table.

vartypes The typecodes for the table fields.

4.6.2 `Table` methods

appendAsRecord(RecordObject) Append the `RecordObject` to the output buffer of the table instance.

> **RecordObject** An instance of the user-defined record class. It has to be a `IsRecord` descendant instance and if not, a ValueError exception is raised.

appendAsTuple(tupleValues) Append the *tupleValues* tuple to the output buffer of the table instance. This method is faster (but also unsafer, because requires user to introduce the values in correct order!) than `appendAsRecord` method.

> **tupleValues** is a tuple that has values for all the user record fields. The user has to provide them in the order determined by alphanumerically sorting the record name fields.

appendAsValues(*values) Append the *values* parameters to the table output buffer. This method is faster (and unsafer, because requires user to introduce the values in correct order) than `appendAsRecord` method. It is similar to the `appendAsTuple` method, but accepts separate parameters as values instead of a monolithic tuple.

> **values** Is a serie of parameters that provides values for all the user record fields. The user has to provide them in the order determined by alphanumerically sorting the record fields.

readAsRecords() Return a record instance from rows in table each time. This method is a *generator*, i.e. it keeps track on the last record returned so that next time it is invoked it returns the next available record. It is slower than `readAsTuples` but in exchange, it returns full-fledged instance records.

readAsTuples() Return a tuple from rows in table each time. This method is a *generator*, i.e. it keeps track on the last record returned so that next time it is invoked it returns the next available record. This method is twice as faster than readAsRecords, but it yields the rows as (alphanumerically orderd) tuples, instead of full-fledged instance records.

flush() Flush the table buffers.

close() Flush the table buffers and close the HDF5 dataset.

4.7 The `Array` class

Represent a Numeric Array in HDF5 file. It provides methods to create new arrays or open existing ones, as well as methods to write/read data and metadata to/from array objects in the file.

All Numeric typecodes are supported except `"F"` and `"D"` which corresponds to complex datatypes. These might be included in short future.

4.7.1 `Array` instance variables

name The node name.

title The node title.

shape tuple with the array shape (in the Numeric fashion).

typecode The typecode of the represented array.

4.7.2 `Array` methods

The methods for this class are very few. Please note that this object has not internal I/O buffers, so there is no need to call flush() method. However, it is included for consistency with `Leaf` nodes.

read() Read the array from disk and return it as a Numeric object. Note that while this method is not called, the actual array data is resident on disk.

flush() Flush the internal buffers. Remember: this is a do-nothing method.

close() Close the array on file.

Appendix A

PyTables Supported Data Types

The supported data types are the same that are supported by the array module in Python, with some additions, which will be briefly discussed shortly. The typecodes for the supported data types are listed on table A.

The additions to the array module typecodes are the 'q', 'Q' and 's'. The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes. The 's' typecode can be preceded by an integer to indicate the maximum length of the string, so '16s' represents a 16-byte string.

Also note that when the 'I' and 'L' codetypes are used in records, Python uses internally Long integers to represent them, that can (or cannot, depending on what you are trying to do) be a source of inefficiency in your code.

Type Code	Description	C Type	Size (in bytes)	Python Counterpart
'c'	8-bit character	char	1	String of lenght 1
'b'	8-bit integer	signed char	1	Integer
'B'	8-bit unsigned integer	unsigned char	1	Integer
'h'	16-bit integer	short	2	Integer
'H'	16-bit unsigned integer	unsigned short	2	Integer
'i'	integer	int	4 or 8	Integer
'I'	unsigned integer	unsigned int	4 or 8	Long
'l'	long integer	long	4 or 8	Integer
'L'	unsigned long integer	unsigned long	4 or 8	Long
'q'	long long integer	long long	8	Long
'Q'	unsigned long long integer	unsigned long long	8	Long
'f'	single-precision float	float	4	Float
'd'	double-precision float	double	8	Float
's'	arbitrary lenght string	char[]	*	String

Table A.1: Data types supported by PyTables

Bibliography

1. *What is HDF5?.* Concise description about HDF5 capabilities and its differences from earlier versions (HDF4). `http://hdf.ncsa.uiuc.edu/whatishdf5.html`
2. *Introduction to HDF5.* Introduction to the HDF5 data model and programming model. `http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.intro.html`
3. *HDF5: High Level APIs.* A set of functions built on top of the basic HDF5 library. `http://hdf.ncsa.uiuc.edu/HDF5/hdf5_hl/doc/`
4. *The HDF5 table programming model.* Examples on using HDF5 tables with the C API. `http://hdf.ncsa.uiuc.edu/HDF5/hdf5_hl/doc/RM_hdf5tb_ex.html`
5. *HL-HDF.* A High Level Interface to the HDF5 File Format. `ftp://ftp.ncsa.uiuc.edu/HDF/HDF5/contrib/hl-hdf5/README.html`
6. *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`. `http://www-106.ibm.com/developerworks/xml/library/xml-matters2/index.html`
7. *gnosis.xml.objectify.* This module is part of the Gnosis utilities, and allows to create a mapping between any XML element to "native" Python objects. `http://gnosis.cx/download/Gnosis_Utils-current.tar.gz`
8. *Pyrex.* A Language for Writing Python Extension Modules. `http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex`
9. *NetCDF (network Common Data Form).* This is an interface for array-oriented data access and a library that provides an implementation of the interface. `http://www.unidata.ucar.edu/packages/netcdf/`
10. *NetCDF module on Scientific Python.* ScientificPython is a collection of Python modules that are useful for scientific computing. Its NetCDF module is a powerful interface for NetCDF data format. `http://starship.python.net/~hinsen/ScientificPython/ScientificPythonManual/`
11. *Numerical Python.* Package to speed-up arithmetic operations on arrays of numbers. `http://www.pfdubois.com/numpy/`
12. *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. `http://stsdas.stsci.edu/numarray/`