Francesc Alted

PyTables User’s Guide

Alted, Francesc:

`PyTables` User's Guide

Day of print: March, 18th

Contents

La sabiduría no vale la pena si no es posible servirse de ella para inventar una nueva manera de preparar los garbanzos.

—Un sabio catalán 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 objects in a hierarchical structure. The foundation of the underlying hierarchical 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, but to provide a flexible, *very Pythonic* tool to deal with (arbitrary) large amounts of data (typically bigger than available memory) in tables and arrays organized in a hierarchical, persistent disk storage.

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 scientific applications, for example) in an efficient manner that reduces demand on CPU time and I/O.

In order to emulate records (that will be mapped to C structs in HDF5) in Python `PyTables` implements a special *metaclass* object with the capability to detect errors in assignments to non-existing fields as well as data type incompabilities. `PyTables` also provides a powerful interface to mine data in table. 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 the name field and types information, like in:

```
class Particle(IsColDescr):
    name      = Col('CharType', 16)  # 16-character String
    idnumber  = Col("UInt64", 1)     # Unsigned long long
    ADCcount  = Col("UInt16", 1)     # Unsigned short integer
    TDCcount  = Col("UInt8", 1)      # unsigned byte
    grid_i    = Col("Int32", 1)      # integer
    grid_j    = Col("Int32", 1)      # integer
    pressure  = Col("Float32", 1)    # float  (single-precision)
    energy    = Col("Float64", 1)    # double (double-precision)
```

then, you should have to pass this class to the table constructor, fill its rows 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 (in C, Fortran, Java or whatever language that has an interface to HDF5).

Next section describes the most interesting capabilities of `PyTables`.

1.1 Main Features

`PyTables` has the next capabilities:

- *Support of table entities:* Allows working with a large number of records, i.e. that don't fit in memory.
- *Support of arrays:* `Numeric` or `numarray` arrays are a very useful complement of tables to keep homogeneous 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 data structure. Access to the file objects is achieved by walking throughout this object tree, and manipulating it.
- *Appendable tables:* It supports adding records to already created tables. This can be done without copying the dataset or redefining its structure, even between different Python sessions.
- *Automatically check for correct field name, and data type:* That reduces programmer mistakes and if `PyTables` does not report an error, you can be more 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.
- *Can read generic HDF5 files:* `PyTables` provides limited support to import generic HDF5 files provided they contain any combination compound type datasets (that will be mapped to tables) or homogeneous datasets (that will be mapped to arrays). However, as these kind of data is the most common to be saved HDF5 format, probably `PyTables` can read more than the 99% of the HDF5 files out there.
- *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).
- *Architecture-independent:* `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 (like a Sparc or MIPS) and read it in other little-endian (like Intel or Alpha) without problems.

`PyTables` take advantage of the powerful object orientation and introspection capabilities offered by Python to bring all those exposed features 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. In order to achieve this, an *object tree* entity is *dynamically* created imitating the HDF5 structure on disk. That way, the access to the HDF5 objects is made by walking throughout this object tree, and, by looking at their *metadata* nodes, you can get a nice picture of what kind data is kept there.

The different nodes in the object tree are instances of `PyTables` classes. There are several types of those classes, but the most important ones are the `Group` and the `Leaf`. `Group` instances (that we will be calling *groups* from now on) are a grouping structure containing instances of zero or more groups or leaves, together with supporting metadata. `Leaf` instances (that will be called *leaves*) are containers for actual data and cannot contain any other instances[1]. The `Table` and `Array` classes are descendants of `Leaf`, and inherits all its properties.

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. In `PyTables` this full path can be specified either as string (like in `'/subgroup2/table3'`) or as a complete object path written in a certain way known as *natural name* schema (like in `file.root.subgroup2.table3`).

[1] Except `Attribute` instances in the short future

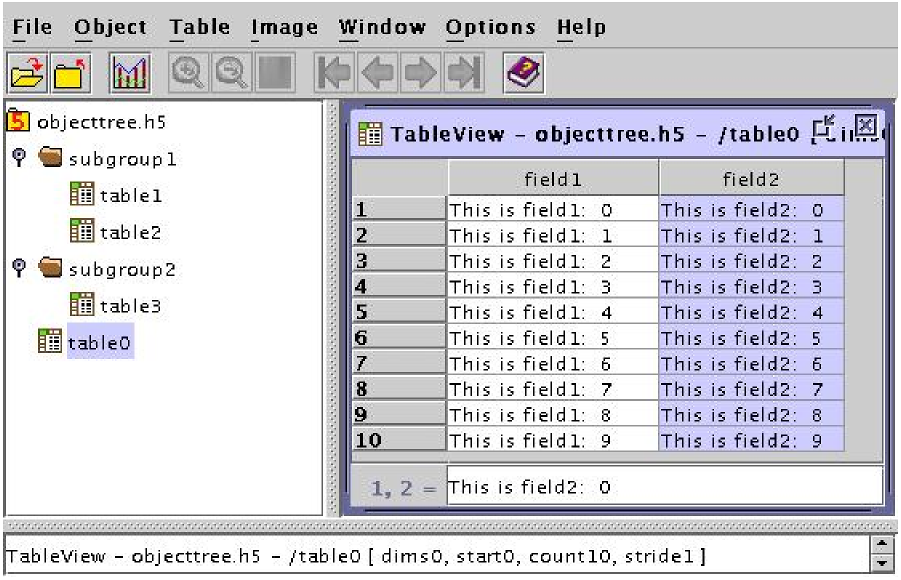

Figure 1.1: An HDF5 example with 2 subgroups and 3 tables.

The support for *natural naming* is a key aspect of `PyTables` and means that the names of instance variables of the node objects are the same as the names of the element's children[2]. This is very *Pythonic* and comfortable in many cases, as you can check in the tutorial section 3.1.6.

You should also note that not all the data present on file is loaded in the object tree, but only the *metadata* (i.e. special data that describes the structure of the actual data). The actual data is not read until you ask for it (by calling a method in a particular node). By making use of the object tree (the metadata) you can get information on the objects on disk such as 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 also traverse the tree in order to search for something 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.

To better understand the dynamic nature of this object tree entity, imagine that we have made a script (in fact, this script actually exists and you can find it in `examples/objecttree.py`; check it out!) that creates a simple `PyTables` file, with the structure that appears in figure 1.1. During creation time, metadata in the object tree is updated in memory while the actual data is being saved on disk and when you close the file the object tree becomes unavailable. But, when you will open again this file the object tree with will be re-constructed in memory from the metadata existent on disk, so that you can work 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 (in fact, this file is the same as that of the figure 1.1). If you are going to become a `PyTables` user, take your time to understand it[3]. That will also make you more proactive by avoiding programming mistakes.

[2] I have got this simple but powerful idea from the excellent `Objectify` module by David Mertz (see references 7 and 8)

[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.

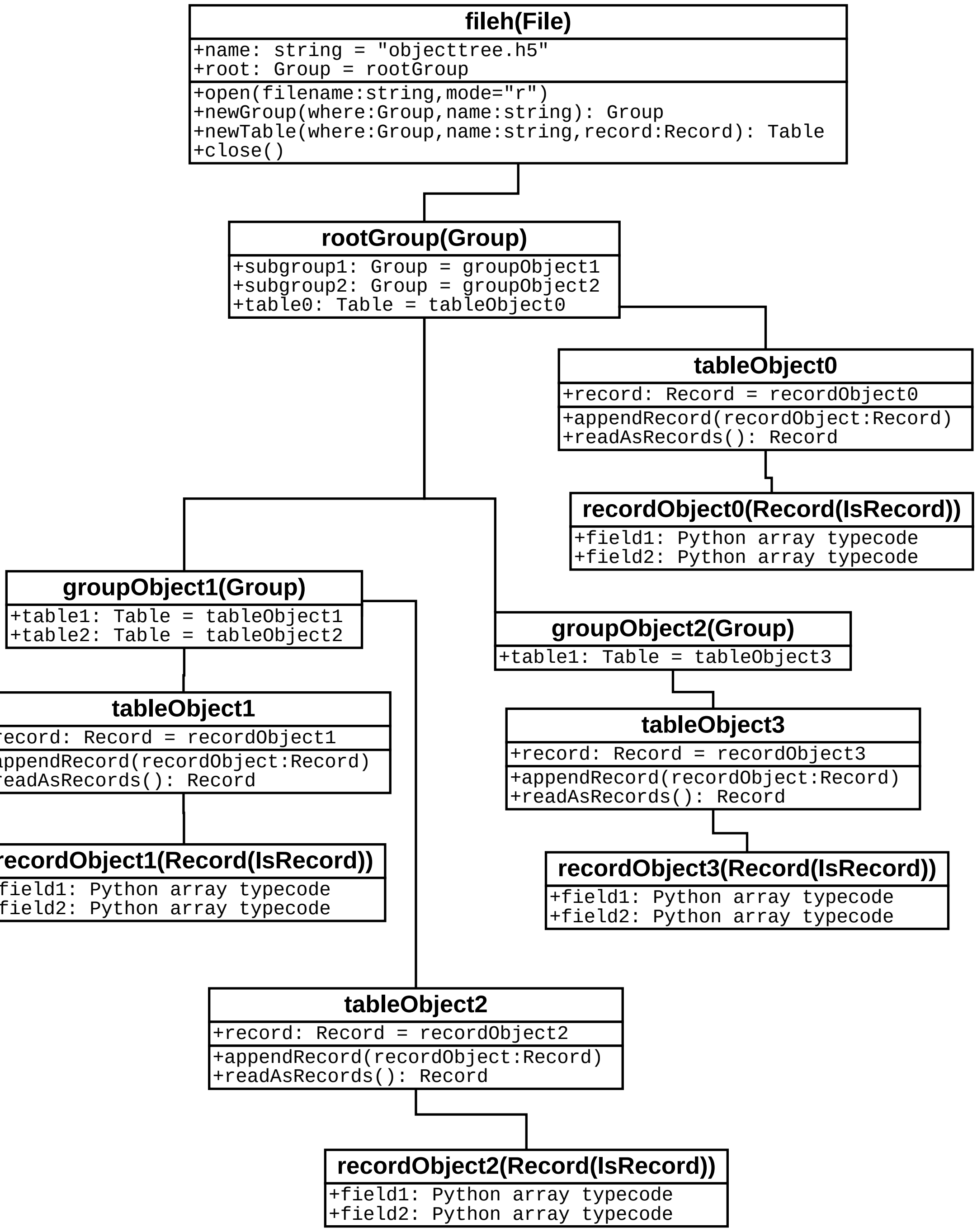

Figure 1.2: An object tree example in `PyTables`.

Chapter 2

Installation

These 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 9) and C language. 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`.

To compile `PyTables` you will need a recent version of `HDF5` (C flavor) library and `numarray` (see reference 13) package. Although you won't need `Numerical Python` in order to compile PyTables, it is supported; you only will need a reasonably recent version of it (>= 21.x). PyTables has been succesfully tested with Numeric 21.3, 22.0 and 23.0. If you have `Numeric` installed, the test driver module will detect it and will run the tests for `Numeric` automatically.

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 `numarray` installed (I'm using `HDF5 1.4.5` and `numarray 0.4` 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` environment 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:

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

   If there is some test that do not pass, please, run the failing test module with all verbosity enabled (flags `-v verbose`), and send back the output to developers.

   If you run into problems because Python can't load hdf5 shared libraries, try to set the `LD_LIBRARY_PATH` environment variable to point to the directory where the libraries are.

4. To install the entire `PyTables` Python package, change back to the root distribution directory and run this command as the root user (remember to add any extra flags needed):

   ```
   python setup.py install
   ```

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

Chapter 3

Usage

This chapter begins with a series of simple, yet comprehensive sections written in a tutorial style that will let you understand the main features that PyTables provide. If during the trip you want more information on some specific instance variable, global function or method, look at the doc strings or go to the library reference in chapter 4. However, if you are reading this in PDF or HTML formats, there should be an hyperlink to its reference near each newly introduced entity. Finally, you can get deeper knowledge of PyTables internals by reading the last section (3.4) in this chapter.

Please, note that throughout this document the terms *column* and *field* will be used interchangeably with the same meaning, and the same goes for the terms *row* and *record*.

3.1 Getting started

In this section, 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 select some data in the table using Python cuts, creating Numerical arrays to keep this selection as separate objects in the 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. You are encouraged to take advantage of that by doing parallel testing and inspecting the created objects (variables, docs, children objects, etc.) during the voyage!.

3.1.1 Importing `tables` objects

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

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

This 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 *
>>>
```

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

If you are going to deal with `Numeric` or `numarray` arrays (and normally, you will) you 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:

```
>>> import tables        # but in this tutorial we use "from tables import *"
>>> from Numeric import *  # or "from numarray 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 that comes from it. You need first to define that table, how many columns it have, which kind of object is each element on the columns, and so on.

Our 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 and we will add two new fields called `grid_i` and `grid_j`. Our instrumentation also can obtain the pressure and energy of this particle that we want to add in the same way. The resolution of pressure-gauge allows us to use simple-precision float which 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 can declare a new `Particle` class that will keep all this info:

```
>>> class Particle(IsColDescr):
...     name      = Col('CharType', 16)  # 16-character String
...     idnumber  = Col("UInt64", 1)     # Unsigned long long
...     ADCcount  = Col("UInt16", 1)     # Unsigned short integer
...     TDCcount  = Col("UInt8", 1)      # unsigned byte
...     grid_i    = Col("Int32", 1)      # integer
...     grid_j    = Col("Int32", 1)      # integer
...     pressure  = Col("Float32", 1)    # float  (single-precision)
...     energy    = Col("Float64", 1)    # 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 we assign a `Col` instance, that takes as arguments the data type and the number of items on each column element (in fact, the `Col()` constructor accepts a few more arguments, see section 4.3 for a detailed descrition). See appendix A for a list of data types supported in `Col` constructors.

From now on, we can use `Particle` instances as a descriptor for our detector data table. We will see how to pass this object to the `Table` constructor. 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` (see 4.1.2) function:

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

This `openFile` is one of the objects imported by the "`from tables import *`", do you remember?. Here, we are telling that we want to create a new file called "`tutorial1.h5`" in "`w`"rite mode and with an descriptive title string ("`Test file`"). This function tries to open the file, and if successful, returns a `File` (see 4.4) 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')
>>>
```

Here, we have taken the `File` instance `h5file` and invoked its `createGroup` method (see 4.4.2), 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` (see 4.5) instance that will be assigned to the `group` variable.

3.1.5 Creating a new table

Let's now create the `Table` (see 4.7) object hanging from the new created group. We do that by calling the `createTable` (see 4.4.2) 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'*. We have passed an instance of `Particle`, the class that we have declared before, as the *description* parameter and finally we have used "*Readout example*" as a `Table` title. With all this information, a new `Table` instance is created and assigned to *table* variable.

If you are getting curious how the object tree looks like at this moment, simply print the name of the `File` instance, *h5file*, and look at their output:

```
>>> print h5file
Filename: tutorial1.h5 'Test file'
/ (Group) 'Test file'
/detector (Group) 'Detector information'
/detector/readout Table(0,) 'Readout example'
>>>
```

As you can see, a dump of the object tree has been shown and it's very easy to visualize the `Group` and `Table` objects we have just created. If you want more information, just type the name of the `File` instance:

```
>>> h5file
>>> h5file
Filename: tutorial1.h5 'Test file'
  mode = 'w'
  trMap = {}
/ (Group) 'Test file'
/detector (Group) 'Detector information'
/detector/readout Table(0,) 'Readout example'
  description = {
    'ADCcount': Col('UInt16', (1,)),
    'TDCcount': Col('UInt8', (1,)),
    'energy': Col('Float64', (1,)),
    'grid_i': Col('Int32', (1,)),
    'grid_j': Col('Int32', (1,)),
    'idnumber': Col('UInt64', (1,)),
    'name': Col('CharType', (16,)),
    'pressure': Col('Float32', (1,)) }
  byteorder = little
>>>
```

where more detailed info is printed on each object on the tree. Pay attention on how `Particle`, our column descriptor class, is printed as part of the *readout* table description information. In general, you can obtain lot of information on the objects and its children by just printing them. That introspection capability is quite powerful, so I recommend you to use it extensively.

Now, time to fill this table with some values. But first, we are going to get a pointer to the `Row` instance of this `table` instance:

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

The `row` attribute of `table` points to the `Row` (see 4.8) instance that will be used to input data rows into the table. We achieve this by just assigning it the values for each row as if it was a dictionary (although it is actually an *extension class*) and using the column names as keys.

Look at how the filling process works like:

```
>>> particle = table.row
>>> 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)
...     # Insert a new particle record
...     particle.append()
...
>>>
```

This code is quite easy to understand. The lines inside the loop just assign values to the different columns in the `particle` row object and then a call to its `append()` (see 4.8) method is made to put this information in the `table` I/O buffer.

After we have pushed all our data, we should flush the I/O buffer for the table if we want to consolidate all this data on disk. We can achieve that by calling the `table.flush()` method.

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

3.1.6 Reading (and selecting) data in table

Ok. We have now our data on disk but to this data be useful we need to access it and select some values we are interested in and located at some specific columns. That's is easy to do:

```
>>> table = h5file.root.detector.readout
>>> pressure = [ x['pressure'] for x in table.iterrows()
...              if x['TDCcount'] > 3 and x['pressure'] < 50 ]
>>> pressure
[16.0, 25.0, 36.0, 49.0]
>>>
```

The first line is only to declare a convenient 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 will certainly do that later on.

The last two lines are a Python comprehensive list. It loops over rows in *table* as they are provided by `table.iterrows()` iterator (see 4.7.2) that returns values until data in table is exhausted. This rows are filtered using the 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 indeed have used a normal `for` loop to do that, but I find comprehension syntax more compact and elegant.

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

```
>>> names = [ x['name'] for x in table.iterrows()
...           if x['TDCcount'] > 3 and x['pressure'] < 50 ]
>>> names
['Particle:      4', 'Particle:      5', 'Particle:      6', 'Particle:      7']
>>>
```

Ok. that's enough for selections. Next section will show you how to save these selections on file.

3.1.7 Creating new array objects

In order to separate the selected data 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 using an absolute path string ("/").

Now, create one `Array` object:

```
>>> h5file.createArray(gcolumns, 'pressure', array(pressure),
...                    "Pressure column selection")
/columns/pressure Array(4,) 'Pressure column selection'
  type = 'Float64'
  itemsize = 8
  flavor = "Numeric"
  byteorder = 'little'
```

We already know the first two parameters of the `createArray` (see 4.4.2) methods (these are the same as the firsts 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 *object* we want to save on disk. In this case, it is a `Numeric` array that is built from the selection lists we created before.

Now, we are going to save the other selection. In this case it's a list of strings, and we want to save this object as is, with no further conversion. Look at how this is done:

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

You see, `createArray()` accepts *names* (which is a regular Python list) as *object* parameter. Actually, it accepts a variety of other regular objects (see 4.4.2). We will check that we can retrieve exactly this same object from disk later on.

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

```
>>> print h5file
Filename: tutorial1.h5 'Test file'
/ (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(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` instance to close the file before exiting Python:

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

With all that, you have created your first `PyTables` file with a table and two arrays. 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 `tutorial1.h5` file:

```
$ h5ls -rd tutorial1.h5
/columns                 Group
/columns/name            Dataset {4}
    Data:
        (0) "Particle:      4", "Particle:      5", "Particle:      6",
        (3) "Particle:      7"
/columns/pressure        Dataset {4}
    Data:
        (0) 16, 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}
```

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

```
$ python dumpFile.py tutorial1.h5
Filename: tutorial1.h5
All objects:
Filename: tutorial1.h5 'Test file'
/ (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(10,) 'Readout example'
```

You can pass the `-v` or `-d` options to `dumpFile.py` if you want more verbosity. Try it!.

3.2 Browsing the *object tree* and more

In this section, we will learn 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.

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 voyage.

3.2.1 Traversing the object tree

First of all, let's open the file we have recently created in last tutorial section, as we will take it as a basis for this section:

```
>>> h5file = openFile("tutorial1.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.

`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: tutorial1.h5 'Test file'
/ (Group) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name Array(4,) 'Name column selection'
/columns/pressure Array(4,) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout Table(10,) 'Readout example'
>>>
```

That's right, it seems that all our objects are there. We can use the `walkGroups` method (see 4.4.2) of `File` class to list all 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. Combining this iterator with the `listNodes()` method, we can do very powerful things. 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,) 'Name column selection'
/columns/pressure Array(4,) 'Pressure column selection'
```

`listNodes()` (see 4.4.2) lists all the nodes hanging from a *group*, and if *classname* keyword is specified, the method will filter all instances which are not descendants of it. We have specified it so as to return only the `Array` instances.

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

As a final example, we will list all the `Leaf` (i.e. `Table` and `Array` instances, see 4.6 for detailed information on `Leaf` class) objects in */detector* group. Check that only one instance of `Table` class (i.e. *readout*), will be selected in this group (as it should be):

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

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

3.2.2 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 have 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: %d" % (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, ':=', table.coltypes[name], table.colshapes[name]
...
   ADCcount := UInt16 (1,)
   TDCcount := UInt8 (1,)
   energy := Float64 (1,)
   grid_i := Int32 (1,)
   grid_j := Int32 (1,)
   idnumber := UInt64 (1,)
   name := CharType (16,)
   pressure := Float32 (1,)
>>>
```

Here, the `name`, `title`, `nrows`, `colnames`, `coltypes` and `colshapes` attributes (see 4.4.1 for a complete attribute list) of `Table` object give us quite a lot of information about actual table data.

In general, you can get up-to-the-minute information about the public objects in PyTables in a interactive way by printing its 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]])
    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)
>>>
```

This is very handy if you don't have this manual at hand. Try yourself with other objects docs!.

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

```
>>> pressureObject = h5file.getNode("/columns", "pressure")
>>> print "Info on the object:", repr(pressureObject)
Info on the object: /columns/pressure Array(4,) 'Pressure column selection'
  type = 'Float64'
  itemsize = 8
  flavor = "Numeric"
  byteorder = 'little'
>>> print "  shape: ==>", pressureObject.shape
  shape: ==> (4,)
>>> print "  title: ==>", pressureObject.title
  title: ==> Pressure column selection
>>> print "  type ==>", pressureObject.type
  type ==> Float64
>>>
```

Observe how we have used the `getNode()` method of `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 to use one or another. `getNode()` has the advantage that it can get a node from the pathname string (like in this example), and, besides, you can force a filter so 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. I suggest to give a try at this powerful combination of natural naming and completion capabilities present on most Python shell. You will see that browsing the object tree in that way is a very pleasant experience (well, as far as this activity can be qualified as *pleasant*).

If you look at the `typecode` attribute of the `pressureObject`, you can certify that this is a "**Float64**" array, and that by looking at their `shape` attribute, it can deduced that the array on disk is unidimensional and has 4 elements. See 4.9.1 or the internal string docs for the complete `Array` attribute list.

3.2.3 Reading actual data from `Array` objects

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

```
>>> pressureArray = pressureObject.read()
>>> nameArray = h5file.root.columns.name.read()
>>> print "pressureArray is an object of type:", type(pressureArray)
pressureArray is an object of type: <type ’array’>
>>> pressureArray
array([ 16.,  25.,  36.,  49.])
>>> print "nameArray is an object of type:", type(nameArray)
nameArray is an object of type: <type ’list’>
>>> nameArray
[’Particle:      4’, ’Particle:      5’, ’Particle:      6’, ’Particle:      7’]
>>>
>>> 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:      4 --> 16.0
Particle:      5 --> 25.0
Particle:      6 --> 36.0
Particle:      7 --> 49.0
>>>
```

You can verify as the `read()` method (see 4.9.2) returns an authentic `Numeric` array looking at the output of the `type()` call. This is because the type of the object saved is kept as an HDF5 attribute (named `FLAVOR`) for this object on disk. This attribute is then read as part of the `Array` metainformation and accessible through the `Array.flavor` instance 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 recover them in its original form afterwards. See section 4.4.2 for a complete list of supported objects.

3.2.4 Appending data to an existing table

To finish this section, 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:

```
>>> 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()
>>>
```

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[1].

If you look carefully at the code you will see that we have used the `table.row` attribute so as to access a table row and fill it up 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.

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

```
>>> 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 |
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 |
```

In figure 3.1 you can see a graphical view of the `PyTables` file we have just created.

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

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

3.3 `PyTables` automatic sanity checks

Now, time for a more real life example (i.e. with errors in code). Here, we will create a couple of groups hanging directly from `root` called `Particles` and `Events`. Then, we will put 3 tables in each group; in `Particles` we will put tables based on `Particle` descriptor and in `Events`, tables based `Event` descriptor.

After that, we will feed the tables with a number of records. Finally, we will read the recently created table `/Events/TEvent3` and select some values from it using a comprehension list.

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

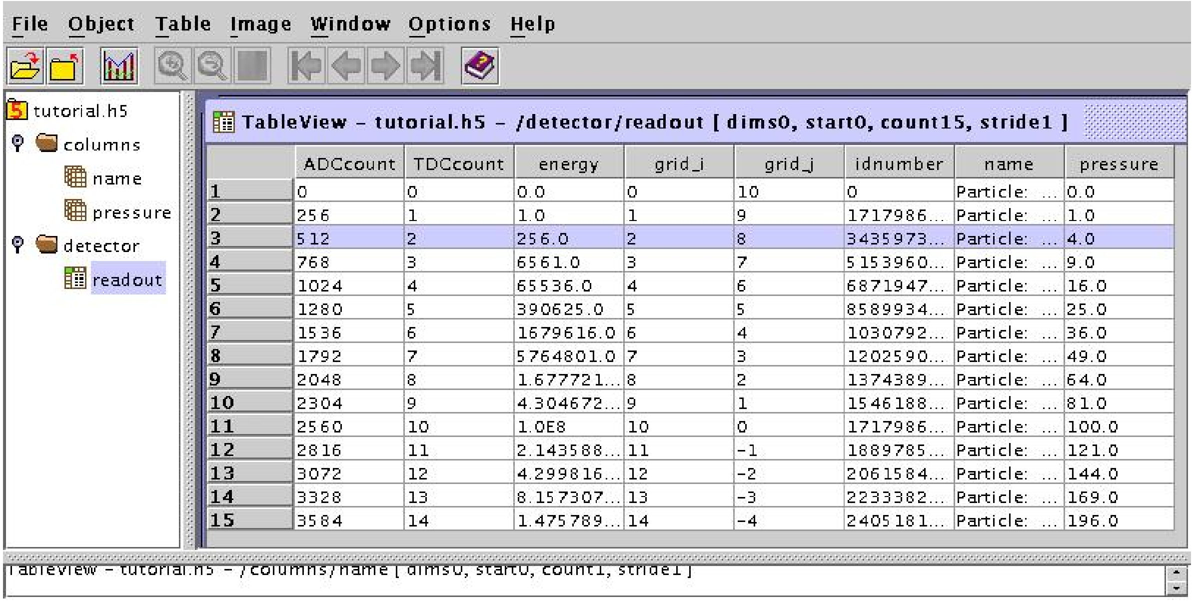

Figure 3.1: The data file after appending some rows.

Look at the next script. It seems to do all of that, but a couple of small bugs will be shown up. Note that this `Particle` class is not directly related with the one defined in last example; this one is simpler.

```
from tables import *
class Particle(IsColDescr):
    name      = Col('CharType', 16)  # 16-character String
    lati      = Col("Int32", 1)      # integer
    longi     = Col("Int32", 1)      # integer
    pressure  = Col("Float32", 1)    # float  (single-precision)
    temperature = Col("Float64", 1)    # double (double-precision)
class Event(IsColDescr):
    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 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 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
        particle['pressure'] = float(i*i)
        particle['temperature'] = float(i**2)
        # This injects the Record values
        particle.append()
    # 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.row
    # 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 % (1<<8)
        event['xcoor'] = float(i**2)
        event['ADCcount'] = i * 2
        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.iterrows()
      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 Field name checking

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

```
Traceback (most recent call last):
  File "tutorial2-bad.py", line 69, in ?
    event['xcoor'] = float(i**2)     # Wrong spelling
  File "/home/falted/PyTables/pytables-0.4/src/hdf5Extension.pyx", line 1078, in __setitem__
    raise AttributeError, "Error accessing \"%s\" attr.\n %s" % \
AttributeError: Error accessing "xcoor" attr.
 Error was: "exceptions.KeyError: xcoor"
```

This error is telling us that we tried 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 very unusual in Python because if you try to assign a value to a non-existent instance variable, a new one is created with that name. Such a feature is not satisfactory when we are dealing with an object that has fixed list of field names. So, a check is made inside PyTables so that if you try to assign a value to a non-existing field a `KeyError` is raised.

3.3.2 Data type checking

Finally, in order to test the type checking, 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:

```
Traceback (most recent call last):
  File "tutorial2-bad.py", line 68, in ?
    event['ADCcount'] = "s"      # Wrong type
  File "/home/falted/PyTables/pytables-0.4/src/hdf5Extension.pyx", line 1078, in __setitem
    raise AttributeError, "Error accessing \"%s\" attr.\n %s" % \
AttributeError: Error accessing "ADCcount" attr.
 Error was: "exceptions.TypeError: NA_setFromPythonScalar: bad value type."
```

that states the kind of error (`TypeError`).

You can admire the structure we have created with this (corrected) script in figure 3.2. As before, you will find this example in source file `tutorial2.py` that is located in the directory `examples`.

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

3.4 Optimization tips

`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.

3.4.1 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 CPU time resources consumer. However, if you are willing to check if compression can help reducing your dataset, keep reading.

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

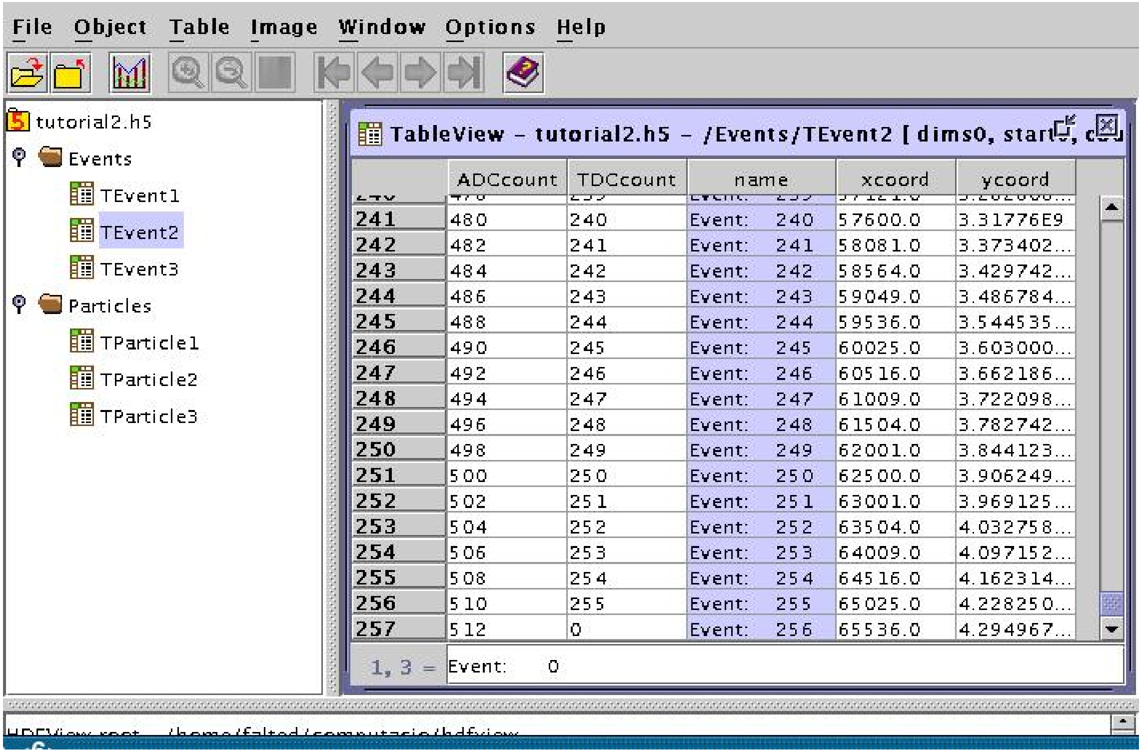

Figure 3.2: Table hierarchy for second example.

they can be referred in the same loop. This process may normally end in a simple, yet powerful manner to process your data (although you should still be careful about what kind of scenarios compression is convenient or not).

The compression library used is the **zlib** (see reference 6), and the compression level that I recommend to use for `Table` objects 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 compress well the possible redundancy on your compound 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 control it by setting the `compress` keyword in the `createTable` method (see 4.4.2). A value of 0 will completely disable compression, 1 is the less CPU time demanding level, while 9 is the maximum level and most CPU intensive.

3.4.2 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 meta data 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 4.4.2).

When your dataset 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 table data. When the dataset 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!.

Chapter 4

Library Reference

`PyTables` implements several classes to represent the different nodes in the object tree. They are named `File`, `Group`, `Leaf`, `Table` and `Array`. Another one is responsible to build record objects from a subclass user declaration, and performs field and type checks; its name is `IsColDescr`. 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 an user supplied file is a `PyTables` or `HDF5` file. These are called `isPyTablesFile` and `isHDF5`. Finally, several first-level 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 first-level 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 might be useful when reporting bugs.

4.1.2 Global functions

openFile(filename, mode='r', title='', trMap={}) Open a `PyTables` 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 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.
- **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 name HDF5 nodes with invalid or reserved words in Python.

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 `IsColDescr` 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 define columns in `Table` objects through the use of classes that inherit properties from it. In order to define such an special class, you have to declare it as descendent from *IsColDescr*, 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 4.3 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.

4.3 The `Col` class

This class is used as a mean to declare the different properties of a column of a table. The only public method accesible is the constructor itself.

Col(dtype="Float64", shape=(1,), dflt=None, pos = None) Define properties for a `Table` column.

dtype The data type for the column. See the appendix A for a relation of data types supported in a `IsColDescr` class declaration.

shape An integer (or, for multidimensional cases, a tuple, although this is not yet supported as for the version 0.4) that specifies the number of *dtype* items for each element (or shape, for multidimensional elements) of this column.

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 *dflt* is not supplied, a appropriate zero value (or *null* string) will be chosen by default.

pos By default, columns are disposed in memory following an alphanumerical 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 wanted disposition.

4.4 The `File` class

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

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

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

4.4.1 `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.

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.

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.

4.4.2 `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 description for this group.

createTable(where, name, ColDescr, title='', compress=0, 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.
>
> **name** The name of the new table.
>
> **ColDescr** An instance of a user-defined class (derived from the `IsColDescr` 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 ?? 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.
>
> **compress** 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.
>
> **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 3.4.2 for a detailed justification of that issue.

createArray(where, name, ArrayObject, title='') Create a new `Array` instance 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.
- **ArrayObject** 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).
- **title** A description for this object.

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'`. 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 rigth 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 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'`. 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 descendents 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 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 in a top to bottom order, and 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 leaves in the object tree.

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

4.5 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 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 a translation map dictionary in the `File.openfile()` method (see section 4.1.2) so as to use non valid Python names as node names in the file.

4.5.1 `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_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.

4.5.2 `Group` methods

Caveat: These 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 ones 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 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.

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

4.6 The `Leaf` class

This is a helper class useful to place common functionality of all Leaf objects. It is also useful for classifying purposes. 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.

The public variables and methods that `Table` and `Array` inherits from `Leaf` are listed below.

4.6.1 `Leaf` instance variables

name The Leaf node name in Python namespace.

hdf5name The Leaf node name in HDF5 namespace.

title The Leaf title.

shape The shape of the associated data in the Leaf.

byteorder The byteorder of the associated data of the Leaf.

4.6.2 `Leaf` instance variables

rename(newname) Change the name of this leaf to *newname*.

remove() Remove this leaf.

getAttr(attrname) Gets the HDF5 attribute *attrname* of this leaf.

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

flush() Flush the leaf buffers.

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

4.7 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 read/write data and metadata 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 4.8). See example ?? on how to use the `Row` interface.

Please note that this object inherits all the public attributes and methods from `Leaf`.

4.7.1 `Table` instance variables

description The metaobject describing this table

row The `Row` instance for this table.

nrows The number of rows in this table.

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).

4.7.2 `Table` methods

iterrows(start=None, stop=None, step=None) Returns an iterator yielding Row instances built from rows in table. This method is actually a *generator*, i.e. it keeps track on the last record returned so that next time it is invoked it returns the next available row. 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.

- **start** Sets the starting row to return data. It accepts negative values meaning that the count starts from the end.
- **stop** Sets the last row to be returned 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 0 means the last row.
- **step** When step is given, it specifies the increment. Negative values are not allowed right now.

read(self, start=None, stop=None, step=None, field=None, flavor=None) Returns the actual data in `Table`. If *field* is not supplied, it returns the data as a `RecArray` object table.

- **start** Sets the starting row to return data. It accepts negative values meaning that the count starts from the end.
- **stop** Sets the last row to be returned 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 0 means the last row.

step When step is given, it specifies the increment. Negative values are not allowed right now.

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` object. *flavor* must have any of the next values: `Numeric`, `Tuple` or `List`.

4.8 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.

4.8.1 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.

4.9 The `Array` class

Represents an array on 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.

Caveat: All `Numeric` and `numarray` typecodes are supported except those that corresponds to complex data types[2]. See `numarray` manual (reference 13) to know more about the supported data types, or see appendix A.

Please note that this object inherits all the public attributes and methods from `Leaf`.

4.9.1 `Array` instance variables

type The type class of the represented array.

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

4.9.2 `Array` methods

Note that, as this object has not internal I/O buffers, there is no point in calling flush() method inherited from `Leaf`.

read() Read the array from disk and return it as a `NumArray` (default) object, or if possible, with the original *flavor* that it was saved. The supported flavors are: `NumArray`, `CharArray`, `Numeric`, `List`, `Tuple`, `String`, `Int` or `Float`. Note that as long as this method is not called, the actual array data is resident on disk, not in memory.

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

Appendix A

Supported data types in tables

`IsColDescr` descendants supports a limited set of data types to define the table fields. This is roughly the same that the set supported by the `numarray` package (see reference 13) in Python, with the exception of the complex datatypes that are not supported yet. The supported set is listed on table A.

Type Code	Description	C Type	Size (in bytes)	Python Counterpart
’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’	long long integer	long long	8	Long
’UInt64’	unsigned long long 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

Table A.1: Data types supported by `IsColDescr` descendants.

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. *zlib.*A Massively Spiffy Yet Delicately Unobtrusive Compression Library. `http://www.gzip.org/zlib/`
7. *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`
8. *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`
9. *Pyrex.* A Language for Writing Python Extension Modules. `http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex`
10. *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/`
11. *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/`
12. *Numerical Python.* Package to speed-up arithmetic operations on arrays of numbers. `http://www.pfdubois.com/numpy/`
13. *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/`