

Francesc Alted

## **PyTables User's Guide**

**Alted, Francesc:**

PyTables User's Guide

All rights reserved.  
© 2002 Francesc Alted

Day of print: October, 8th



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Features . . . . .	1
1.2	The object tree . . . . .	2
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Usage</b>	<b>7</b>
3.1	A first example . . . . .	7
3.2	A somewhat more complex exercise . . . . .	8
<b>4</b>	<b>Library Reference</b>	<b>13</b>
4.1	The <i>File</i> class. . . . .	13
4.2	The <i>IsRecord</i> class. . . . .	14
<b>A</b>	<b>PyTables Supported Data Types</b>	<b>15</b>



# Chapter 1

## Introduction

PyTables is a Python package that allows dealing with HDF5 (see reference 1) tables. In this document, the term *table* means exactly the same than in HDF5 sense (see reference 3):

"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."

Records in tables are also known, in the HDF5 naming scheme, as *compound* data types. So, you can define arbitrary records in Python, like for example:

```
class Particle(IsRecord):
    name          = '16s'   # 16-character String
    lati          = 'i'     # integer
    longi         = 'i'     # integer
    pressure      = 'f'     # float   (single-precision)
    temperature   = 'd'     # double  (double-precision)
```

fill it with your values, and save (large) collections of them in a file. Then, this data can be retrieved and post-processed quite easily with PyTables or another HDF5 application.

You probably noted that the terms "fixed-length" and strict "data types" present in the table definition seems to be strange concepts for an interpreted language like Python, but supporting them is fundamental when we want to save *\*lots\** of data (mainly for scientific applications, but not only that), if we want to do that in a efficient (both in terms of CPU and I/O requirements) way. PyTables allows that.

### 1.1 Features

PyTables has the next features:

- *Support of HDF5 table entities:* Allows working with large number of records that don't fit in memory.
- *Supports a hierarchical data model:* So, you can structure very clearly all your data. This is also very important when dealing with XML data. Pytables builds up an object tree in memory that replicates the HDF5 structure. That way, the access to the HDF5 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:* So you can be confident that if PyTables doesn't report an error, you can be confident that your data is probably ok.

- *Support of files bigger than 2 GB:* This is because HDF5 already can do that (if your platform supports the C long long integer, or, on Windows, `__int64`).
- *Data compression:* It supports data compression (through the use of the `zlib` library) out of the box. This becomes important when you have repetitive data patterns and don't have time for searching an optimized way to save them.
- *Big-Endian/Low-Endian safety:* `PyTables` has been coded (as HDF5 is) to care with little-endian/big-endian byte orderings. So, in principle, you can write a file in a big-endian machine and read it in other little-endian without problems<sup>1</sup>.

It's important to stress that `PyTables` doesn't support every functionality present on HDF5, so you would be able to create HDF5 files with `PyTables`, and read them with other HDF5 generic tools (like `h5dump`), but you can't hope `PyTables` can read every HDF5 file created with tools different than `PyTables`, as it supports only table objects. However, I'm pondering to extend the support to other objects too, like `NumArray` (see reference 12) entities.

In the same sense, it should be noted that `PyTables` do not pretend to be merely a wrapper of `HDF5_HL` library (don't confuse with `HL-HDF5`, the Swedish Meteorological and Hydrological Institute effort to provide another high Level interface to HDF5; see reference 5), but to provide a flexible tool to deal with HDF5 files. This is achieved by taking advantage of the powerful object orientation and introspection capabilities offered by Python.

## 1.2 The object tree

`PyTables` take advantage of the HDF5 hierarchical model to allow tables to be managed in a tree-like structure. It achieves that by 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. I've got this powerful idea from the excellent `Objectify` module by David Mertz (see references 6 and 7).

You should note that not all the data present on the HDF5 file is loaded in `PyTables` tree, but only the *metadata* (i.e. data that actually describes the structure of the real data). The actual access to the real data is provided through the methods of those objects.

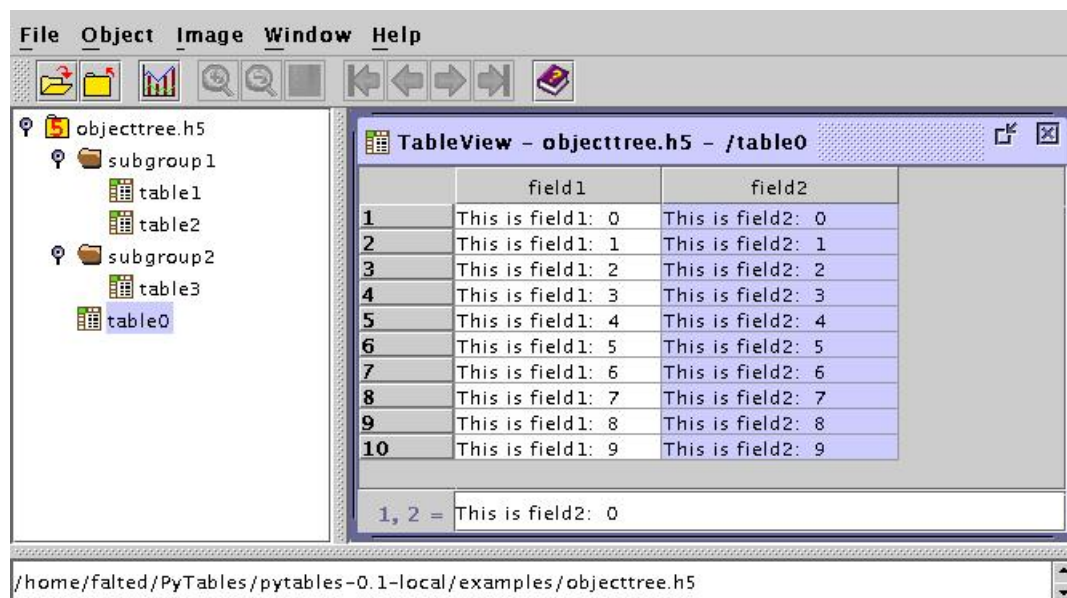
For example, imagine we have made a script (in fact, this script exists; its name is `objecttree.py` and you can find it in the `examples/` directory) that has created a simple HDF5 file, with the structure that appears in figure 1.1 (we have used the java program `hdfview` to obtain this image). If you re-open again this file (in read only mode, for example), the object tree with the HDF5 metadata will be constructed from this hierarchy.

In figure 1.2 you can see an example of the object tree created by reading an HDF5 file (previously written with `PyTables`). It's important that you get familiar with this diagram to better understand how to work with `PyTables`. If you are going to be a `PyTables` user, take your time to study and understand it (bear in mind, however, that this diagram is not a standard UML class diagram; I've used a UML tool to draw it, that's all).

It's important to bear all this in mind while you are working with `PyTables`, because it will ease your work and will make you more proactive by avoiding programming mistakes.

---

<sup>1</sup> Well, I didn't actually test that in real world, but if you do, please, tell me.



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



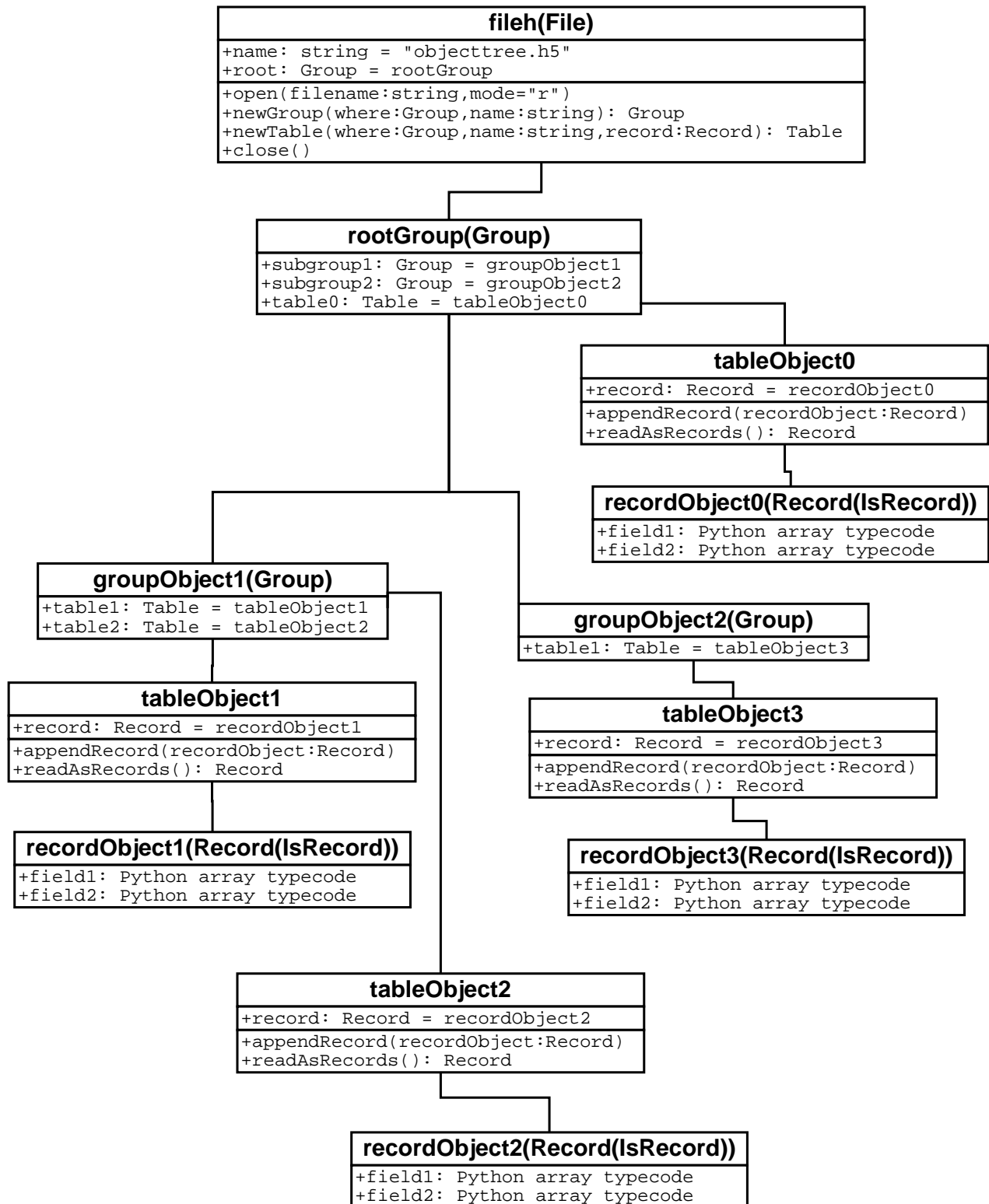


Figure 1.2: An object tree example in PyTables.

## Chapter 2

# Installation

This are instructions for Unix/Linux system. If you are using Windows, and get the library working, 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. But if you want to do that, merely replace `setup.py` script in these instructions by `setup-pyrex.py`.

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

1. First, make sure that you have `hdf5 1.4.x` and `hdf5_hl` libraries installed (I'm using `hdf5 1.4.4` and `hdf5_hl beta2` currently). If not, you can find them at <http://hdf.ncsa.uiuc.edu/HDF5>; compile/install them.

`setup.py` will detect these libraries and include files under either `/usr` or `/usr/local`; this will catch installations from RPMs and most hand installations under Unix. If `setup.py` can't find your `libhdf5` and `libhdf5_hl` or if you have several versions installed and wants 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
```

The libraries can installed anywhere on the filesystem, but remember to always place them together. For example, if `libhdf5.so` is installed in `/usr/lib`, so does `hdf5_hl.so`. The same applies to the headers.

If your HDF5 libs were built as shared libraries, and if these shared libraries are not on 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 to be sure.

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

```
--libs="-lhdf5-1.4.6 -lhdf5_hl-beta2"
--libs="-lhdf5-1.4.6 -lhdf5_hl-beta2 -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, it 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 bash or compatible):

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

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
```

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 some program examples.

## Chapter 3

# Usage

### 3.1 A first example

Let's start by showing a simple example. For simplicity and direct comparison, I'll choose the same that is exposed in an HDF5\_HL example (see reference 4).

So, we want to create a table whose records are particle properties. Each particle (record) has a name, a position (specified by latitude and longitude), pressure and temperature.

We start by define this record in PyTables by declaring a subclass of `IsRecord`. But first, the necessary imports:

```
from tables import File, IsRecord
class Particle(IsRecord):
    name          = '16s' # 16-character String
    lati          = 'i'   # integer
    longi         = 'i'   # integer
    pressure      = 'f'   # float (single-precision)
    temperature   = 'd'   # double (double-precision)
```

As you see, we define the `Particle` class as a subclass of `IsRecord` (which is actually a *metaclass*, but this is not important now). The name of each `Particle` attribute will be the name of the record field and its value will become its data type. '16s' typecode means a 16-character string, 'i' an integer, 'd' a double, and so on. For a complete list of data types supported see table A.1.

Now, we open an HDF5 file in write mode:

```
fileh = File(filename = "example1.h5", mode = "w")
```

and get the object which is the root directory in HDF5 hierarchy:

```
group = fileh.getRootGroup()
```

then, create a new table object

```
table = fileh.newTable(group, 'table', Particle(), "Title example")
```

get the the `Particle` instance associated with the table

```
particle = fileh.getRecordObject(table)
```

and fill the table with 10 particles

```
for i in xrange(10):
    # First, assign the values to the Particle record
    particle.name = '%16d' % i
```

```
particle.lati = i
particle.longi = i
particle.pressure = float(i)
particle.temperature = float(i)
# This injects the Particle values
fileh.appendRecord(table, particle)
```

and finally, close the file:

```
fileh.close()
```

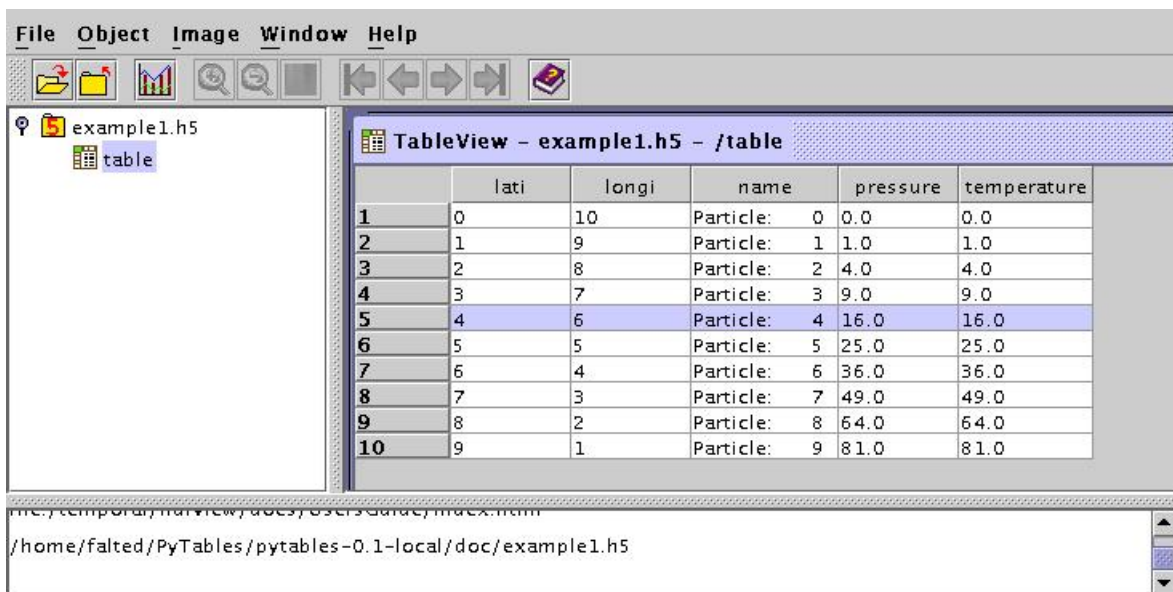
That's it!. We can see here the complete example for a better inspection, with a few additional comments:

```
from tables import File, IsRecord
class Particle(IsRecord):
    name          = '16s'  # 16-character String
    lati          = 'i'    # integer
    longi         = 'i'    # integer
    pressure      = 'f'    # float (single-precision)
    temperature   = 'd'    # double (double-precision)
# Open a file in "w"rite mode
fileh = File(name = "example1.h5", mode = "w")
# Get the HDF5 root group
root = fileh.getRootGroup()
# Create a new table
table = fileh.newTable(root, 'table', Particle(), "Title example")
#print "Table name ==>", table._v_name
# Get the record object associated with the table: all three ways are valid
#particle = table.record
particle = fileh.getRecordObject(table) # This is really an accessor
#particle = fileh.getRecordObject("/table")
# Fill the table with 10 particles
for i in xrange(10):
    # 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. Both ways do that.
    #table.appendRecord(particle)
    fileh.appendRecord(table, particle)
# Finally, close the file
fileh.close()
```

In figure 3.1 you can see the table we have created in this example. You will find in the directory examples the working version of the code (source file `example1.py`).

## 3.2 A somewhat more complex exercise

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"



	lati	longi	name	pressure	temperature
1	0	10	Particle: 0	0.0	0.0
2	1	9	Particle: 1	1.0	1.0
3	2	8	Particle: 2	4.0	4.0
4	3	7	Particle: 3	9.0	9.0
5	4	6	Particle: 4	16.0	16.0
6	5	5	Particle: 5	25.0	25.0
7	6	4	Particle: 6	36.0	36.0
8	7	3	Particle: 7	49.0	49.0
9	8	2	Particle: 8	64.0	64.0
10	9	1	Particle: 9	81.0	81.0

Figure 3.1: A simple table in HDF5.

number) entries each. Finally, we will read the recently created table `/Events/TEvent3` and select some values from it using a comprehension list.

Lets go,

```
from tables import File, IsRecord
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 = File(name = "example2.h5", mode = "w")
# Get the HDF5 root group
root = fileh.getRootGroup()
# Create the groups:
for groupname in ("Particles", "Events"):
    group = fileh.newGroup(root, groupname)
# Now, create and fill the tables in Particles group
gparticles = fileh.getNode("/Particles")
# You can achieve the same result with the next notation
# (it can be convenient and more intuitive in some contexts)
#gparticles = root.Particles
# Create 3 new tables
for tablename in ("TParticle1", "TParticle2", "TParticle3"):
```

```
# Create a table
table = fileh.newTable("/Particles", tablename, Particle(),
                      "Particles: "+tablename)
# Get the record object associated with the table:
particle = fileh.getRecordObject(table)
# 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
    fileh.appendRecord(table, particle)
# Flush the table buffers
fileh.flushTable(table)
# Now, go for Events:
for tablename in ("TEvent1", "TEvent2", "TEvent3"):
    # Create a table. Look carefully at how we reference the Events group!.
    table = fileh.newTable(root.Events, tablename, Event(),
                          "Events: "+tablename)
    # Get the record object associated with the table:
    event = table.record
    # Fill the table with 10 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
        fileh.appendRecord(table, event)
    # Flush the buffers
    fileh.flushTable(table)
# Read the records from table "/Events/TEvent3" and select some
e = [ p.TDCcount for p in fileh.readRecords("/Events/TEvent3")
      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()
```

Throughout the comments, you can see that PyTables let's you do things in, generally, more than one way. I don't know if that's good or not, but I'm afraid it is not. This is in part due to the fact that PyTables is in first stages of development, and probably as the API matures, there will be less choices.

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 'xcoord'

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). So we correct this in the source, and run it again.

And 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  $0 \leq \text{TDCcount} \leq 255$ . 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. As before, you will find in the directory examples the working version of the code (source file example2.py).

Finally, admire the structure we have created in figure 3.2.

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



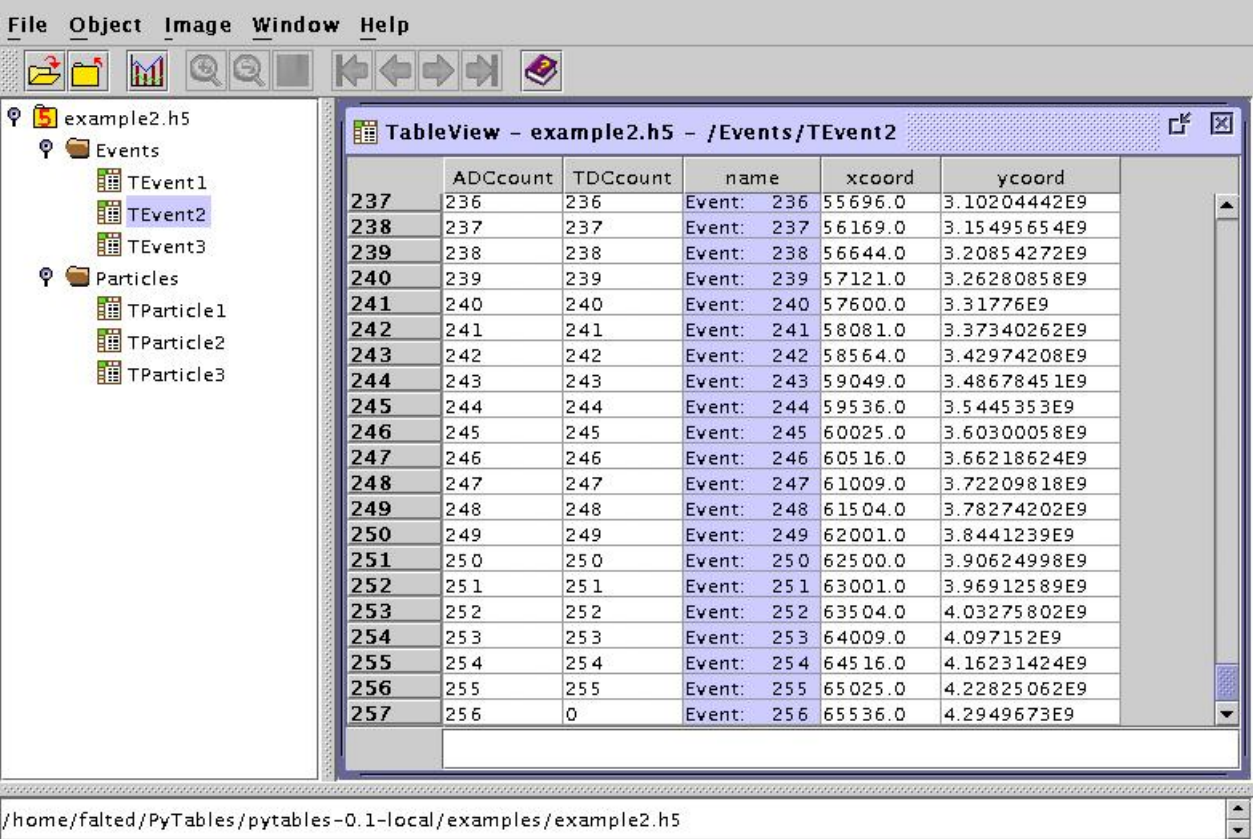


Figure 3.2: Tables structured in a hierarchical order.

## Chapter 4

# Library Reference

This package implements an important class to deal with HDF5 files, called `File` and another one to help defining records, with field, type and range checks, which is called `IsRecord`. There exists other important classes called `Group` and `Table` which do their work silently behind the scenes. The user has to be aware of its existence, but generally speaking, they won't need to call their methods explicitly.

### 4.1 The *File* class.

The `File` class hosts the most part of `PyTables` user interface. It is in charge of create, open, flush and close the HDF5 files. In addition it provides accessors to functionality present in `Group` and `Table` classes.

This class defines the next methods<sup>1</sup>:

**File(filename, mode="r"):** This is the constructor and opens an HDF5 file. The supported access modes are: "r" means read-only; no data can be modified. "w" means write; a new file is created, an existing file with the same name is deleted. "a" means append (in analogy with serial files); 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.

**getRootGroup():** Returns a `Group` instance that will act as the root group in the hierarchical tree. If file is opened in "r", "r+" or "a" mode, and the file already exists, this method dynamically builds a python object tree emulating the structure present on file. It *must* be always called after the `File` object is constructed.

**newTable(when, name, tableTitle = "", compress = 1, expectedrows = 10000):** Returns a new `Table` instance with name *name* in *when* location. *when* parameter can be a path string, or another group instance. Other optional parameters are: *tableTitle* sets a `TITLE` attribute on the HDF5 table entity. *compress* is a boolean option and specifies if data compression will be enabled or not. *expectedrows* is an user estimate about the number of records that will be on table. This parameter is used to set important internal parameters, as buffer size or HDF5 chunk size. If not provided, the default value is appropriate to tables until 100 KB in size. If you plan to save bigger tables by providing a guess to `PyTables` will optimize the HDF5 B-Tree creation and management process time and memory used.

**newGroup(when, name):** Returns a new `Group` instance with name *name* in *when* location. *when* parameter can be a path (for example `"/Particles/TParticle1"` string, or another `Group` instance.

**getNode(when):** Returns the object node (`Group` or `Leave`) in *when* location. *when* can be a path string, `Group` instance or a `Table` instance.

---

<sup>1</sup> On the following, the term `Leaf` will refer to a `Table` instance. Right now the only supported `Leaf` object is `Table`, but that will change in the short future.

**getGroup(where):** Returns the object group in *where* location. *where* can be a path string or a Group instance. If *where* doesn't point to a Group, a ValueError error is raised.

**getTable(where):** Returns the object table in *where* location. *where* can be a path string or a Table instance. If *where* doesn't point to a Table, a ValueError error is raised.

**listNodes(where):** Returns all the object nodes (groups or tables) hanging from *where*. *where* can be a path string or group instance.

**listGroups(where):** Returns all the groups hanging from *where*. *where* can be a path string or group instance.

**listLeaves(where):** Returns all the Leaves objects hanging from *where*. *where* can be a path string or group instance.

**walkGroups(where):** Recursively obtains groups (not leaves) hanging from *where*.

**getRecordObject(table):** Returns the record object associated with the *table*. *table* can be a path string or table instance.

**appendRecord(table, record):** Append the *record* object to the *table* output buffer. *table* can be a path string or table instance.

**readRecords(table):** Generator that returns a Record instance from a *table* object each time it is called. *table* can be a path string or table instance.

**flushTable(table):** Flush the table object to disk. *table* can be a path string or table instance.

**flush():** Flush the buffers for all the objects on the HDF5 file tree.

**close():** Flush all the objects in HDF5 file and close the 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 them 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 your 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.1 for a relation of data types supported in a *IsRecord* class declaration.

## 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.1.

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
<code>'c'</code>	8-bit character	<code>char</code>	1	String of length 1
<code>'b'</code>	8-bit integer	<code>signed char</code>	1	Integer
<code>'B'</code>	8-bit unsigned integer	<code>unsigned char</code>	1	Integer
<code>'h'</code>	16-bit integer	<code>short</code>	2	Integer
<code>'H'</code>	16-bit unsigned integer	<code>unsigned short</code>	2	Integer
<code>'i'</code>	integer	<code>int</code>	4 or 8	Integer
<code>'I'</code>	unsigned integer	<code>unsigned int</code>	4 or 8	Long
<code>'l'</code>	long integer	<code>long</code>	4 or 8	Integer
<code>'L'</code>	unsigned long integer	<code>unsigned long</code>	4 or 8	Long
<code>'q'</code>	long long integer	<code>long long</code>	8	Long
<code>'Q'</code>	unsigned long long integer	<code>unsigned long long</code>	8	Long
<code>'f'</code>	single-precision float	<code>float</code>	4	Float
<code>'d'</code>	double-precision float	<code>double</code>	8	Float
<code>'s'</code>	arbitrary length string	<code>char[]</code>	*	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/](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](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](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/Scientific\\_24.html](http://starship.python.net/~hinsen/ScientificPython/ScientificPythonManual/Scientific_24.html)
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/>