

Francesc Alted

PyTables User's Guide

Alted, Francesc:

PyTables User's Guide

All rights reserved.

© 2002 Francesc Alted

Day of print: October, 8th

Copyright (c) 2002 Francesc Alted

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Copyright Notice and Statement for NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities Copyright 1998, 1999, 2000, 2001, 2002 by the Board of Trustees of the University of Illinois All rights reserved.

Contributors: National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC), Lawrence Livermore National Laboratory (LLNL), Sandia National Laboratories (SNL), Los Alamos National Laboratory (LANL), Jean-loup Gailly and Mark Adler (gzip library).

Redistribution and use in source and binary forms, with or without modification, are permitted for any purpose (including commercial purposes) provided that the following conditions are met:

Put here the conditions....

DISCLAIMER: This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately- owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Contents

1	Introduction	1
1.1	Features	1
1.2	The object tree	2
2	Installation	5
3	Usage	7
3.1	A first example	7
3.2	A somewhat more complex exercise	8
4	Library Reference	13
4.1	tables Variables and Functions	13
4.1.1	Global Variables	13
4.1.2	Global Functions	13
4.2	The IsRecord class	14
4.3	The File class	14
4.3.1	File instance variables	14
4.3.2	File methods	14
4.4	The Group class	16
4.4.1	Group class variables	16
4.4.2	Group instance variables	16
4.4.3	Group methods	16
4.5	The Leaf class	17
4.6	The Table class	17
4.6.1	Table instance variables	17
4.6.2	Table methods	17
4.7	The Array class	18
4.7.1	Array instance variables	18
4.7.2	Array methods	18
A	PyTables Supported Data Types	19

Chapter 1

Introduction

The goal of PyTables is to enable the end user to manipulate easily scientific data **tables** and *Numerical Python* objects in a hierarchical structure. The foundation of the underlying 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.

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

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

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

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

1.1 Features

PyTables has the next capabilities:

- *Support of table entities*: Allows working with large number of records that don't fit in memory.
- *Support of Numerical Python arrays*: Numeric arrays are a very useful complement of tables to keep 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 structure and the access to the file objects is made by walking throughout the `PyTables` object tree, and manipulating them.
- *Incremental I/O:* It supports adding records to already created tables. So you won't need to book large amounts of memory to fill the entire table and then save it to disk but you can do that incrementally, even between different Python sessions.
- *Allows field name, data type and range checking:* So you can be confident that if `PyTables` does not report an error, you can be confident that your data is probably ok.
- *Support of files bigger than 2 GB:* The underlying HDF5 library already can do that (if your platform supports the C long long integer, or, on Windows, `__int64`), so `PyTables` automatically inherits get this benefit.
- *Data compression:* It supports data compression (through the use of the `zlib` library) out of the box. This become important when you have repetitive data patterns and don't want to loose your time searching for an optimized way to save them (i.e. it saves you data organization analysis time).
- *Big-Endian/Low-Endian safety:* `PyTables` has been 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 problems¹.

Finally, it should noted that `PyTables` is not intended 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 large amounts of data (i.e., typically bigger than available memory) in tables (heterogeneous data types) and arrays (homogeneous data types) organized in a hierarchical, persistent disk storage. `PyTables` take advantage of the powerful object orientation and introspection capabilities offered by Python to present all this power to the user in a friendly manner.

1.2 The object tree

The hierarchical model of the underlying HDF5 library allows `PyTables` to manage tables and arrays in a tree-like structure. This is achieved by *dynamically* creating an object tree imitating the HDF5 structure on disk. That way, the access to the HDF5 objects is made by walking throughout the `PyTables` object tree, and manipulating them. A key aspect of `PyTables` is that for accessing to the different nodes on the object tree a **natural naming** schema is used, i.e. the attributes of the objects that represent HDF5 elements are the same as the names of the element's children². See the Chapter 3 for a more detailed explanation.

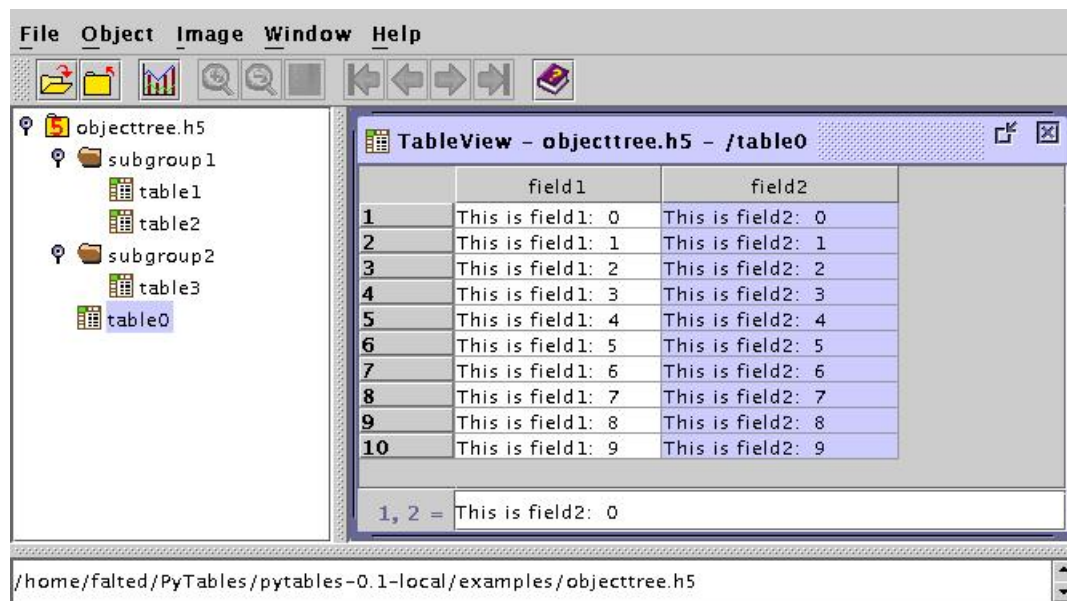
You should note that not all the data present on file is loaded in `PyTables` tree, but only the *metadata* (i.e. data that actually describes the structure of the real data). The actual access to the real data is provided through the methods of those objects.

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

It is important to stress that actual data is not read until you ask for it in a particular node, but by making use of the object tree (the metadata) you can get information on the objects on disk, for example, table names, title, name fields, data types in fields, number of records, or, in the case of arrays, shapes, typecode, and so on. You can traverse the tree with the supplied methods, and when you find the data you want you can read

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

² I've taken this simple but powerful idea from the excellent `Objectify` module by David Mertz (see references 6 and 7)



The screenshot shows the PyTables GUI interface. On the left, the 'Object Tree' pane displays the hierarchy of the 'objecttree.h5' file: a root node 'objecttree.h5' containing two subgroups, 'subgroup1' and 'subgroup2'. 'subgroup1' contains 'table1' and 'table2'. 'subgroup2' contains 'table3' and 'table0'. 'table0' is selected. The main pane, titled 'TableView - objecttree.h5 - /table0', displays a table with 10 rows and 2 columns: 'field1' and 'field2'. The data is as follows:

	field1	field2
1	This is field1: 0	This is field2: 0
2	This is field1: 1	This is field2: 1
3	This is field1: 2	This is field2: 2
4	This is field1: 3	This is field2: 3
5	This is field1: 4	This is field2: 4
6	This is field1: 5	This is field2: 5
7	This is field1: 6	This is field2: 6
8	This is field1: 7	This is field2: 7
9	This is field1: 8	This is field2: 8
10	This is field1: 9	This is field2: 9

Below the table, there is a summary row: '1, 2 = This is field2: 0'. The status bar at the bottom shows the file path: '/home/faltd/PyTables/pytables-0.1-local/examples/objecttree.h5'.

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

and process it. In some sense, you can think of PyTables to provide the introspection capabilities of Python objects, but applied to the persistent storage of large amounts of data.

In figure 1.2 you can see an example of the object tree created by reading a PyTables file. If you are going to be a PyTables user, take your time to understand it³. That will also make you more proactive by avoiding programming mistakes.

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

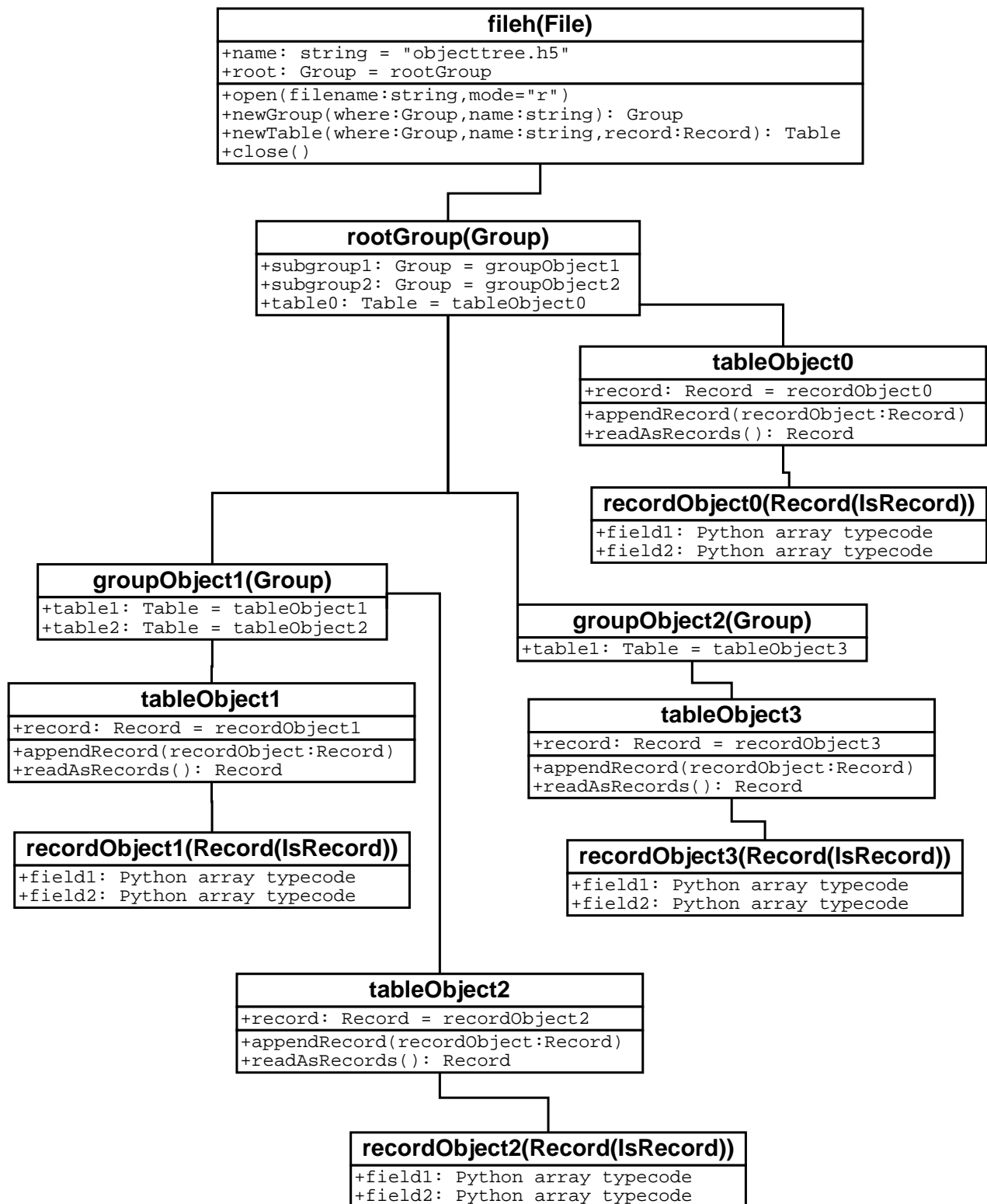


Figure 1.2: An object tree example in PyTables.

Chapter 2

Installation

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

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

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

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

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

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

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

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

```
python setup.py install
```

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

Chapter 3

Usage

3.1 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.

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.

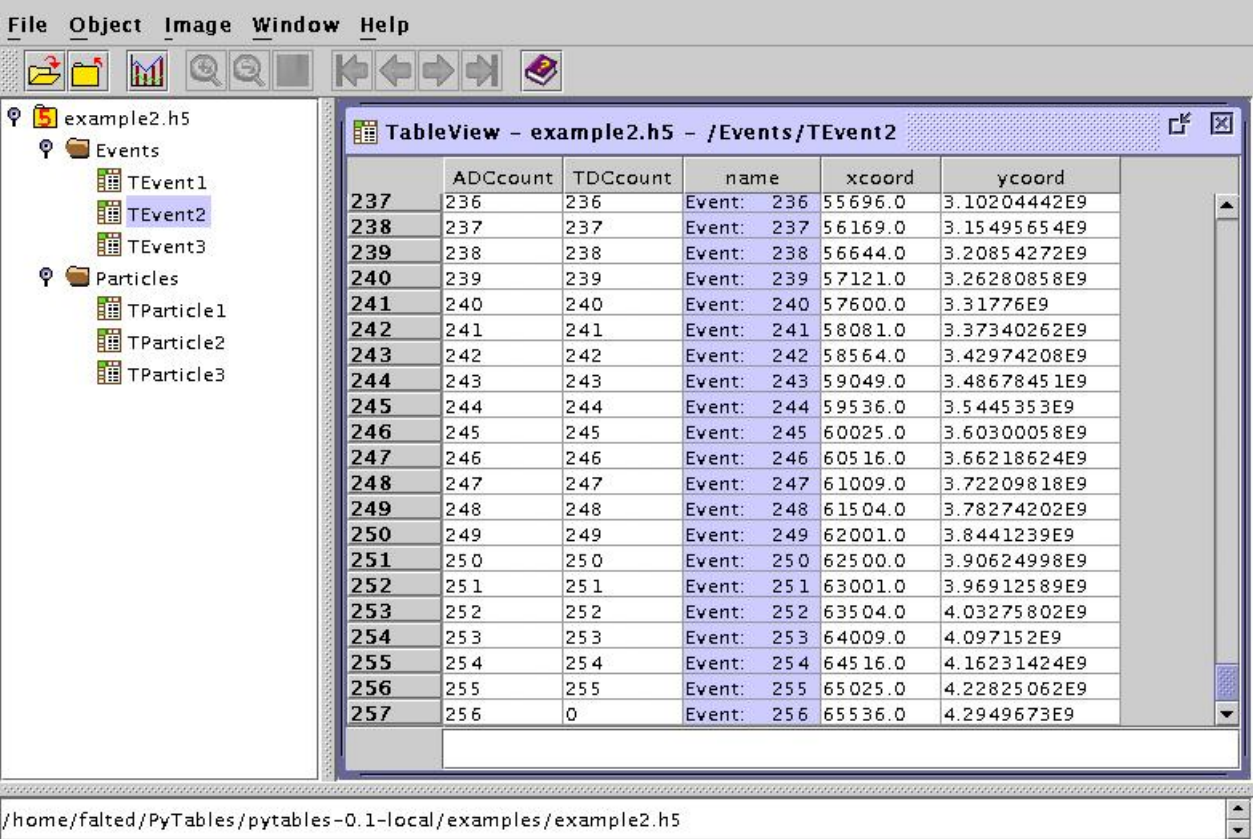


Figure 3.2: Tables structured in a hierarchical order.

Chapter 4

Library Reference

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

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

4.1 tables Variables and Functions

4.1.1 Global Variables

__version__ The PyTables version number.

HDF5Version The underlying HDF5 library version number.

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

4.1.2 Global Functions

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

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

mode: The mode to open the file. It can be one of the following:

'r': read-only; no data can be modified.

'w': write; a new file is created (an existing file with the same name is deleted).

'a': append; an existing file is opened for reading and writing, and if the file does not exist it is created.

'r+': is similar to `'a'`, but the file must already exist.

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

isHDF5(filename) Determines whether filename is in the HDF5 format. When successful, returns a positive value, for `TRUE`, or 0 (zero), for `FALSE`. Otherwise returns a negative value. To this function to work, it needs a closed file.

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

4.2 The IsRecord class

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

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

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

4.3 The File class

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

Next, we will discuss the attributes and methods for `File` class¹.

4.3.1 File instance variables

filename Filename opened.

mode Mode in which the filename was opened.

title The title of the root group in file.

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

4.3.2 File methods

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

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

name The name of the new group.

title A description for this group.

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

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

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

name The name of the new table.

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

title A description for this table.

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

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

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

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

name The name of the new array.

NumericObject The Numeric array to be saved.

title A description for this table.

getNode(where, name='', classname='') Returns the object node *name* under *where* location

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

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

classname If supplied, returns only an instance of this class name. Allowed names in *classname* are: `'Group'`, `'Leaf'`, `'Table'` and `'Array'`.

listNodes(where, classname='') Returns a list with all the object nodes (Group or Leaf) hanging from *where*. The list is alphanumerically sorted by node name.

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

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

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

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

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

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

4.4 The Group class

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

Working with groups and leaves is similar in many ways to working with directories and files, respectively, in a Unix filesystem. As with Unix directories and files, objects in the object tree are often described by giving their full (or absolute) path names. This full path can be specified either as string (like in `'/group1/group2'`) or as a complete object path written in the Pythonic fashion known as *natural name* schema (like in `file.root.group1.group2`) and discussed in the section 1.2.

A collateral effect of the *natural naming* schema is that you must be aware when assigning a new attribute to a Group object to no collide with existing children node names. For this reason and to not pollute the children namespace, it is explicitly forbidden to assign "normal" attributes to Group instances, and the only ones allowed must start with `"_c_"` (for class variables), `"_f_"` (for methods) or `"_v_"` (for instance variables) prefixes. Any attempt to assign a new attribute that does not starts with these prefixes, will raise a `NameError` exception.

4.4.1 Group class variables

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

`_c_objgroups` Dictionary with all object groups on tree.

`_c_objleaves` Dictionary with all object leaves on tree.

4.4.2 Group instance variables

`_v_title` A description for this group.

`_v_name` The name of this group.

`_v_pathname` A string representation of the group location in tree.

`_v_parent` The parent Group instance.

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

`_v_objgroups` Dictionary with all object groups hanging from this instance.

`_v_objleaves` Dictionary with all object leaves hanging from this instance.

4.4.3 Group methods

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

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

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

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

4.5 The Leaf class

This is a helper class useful to place common functionality of all Leaf objects. A Leaf object is an end-node, that is, a node that can hang directly from a group object, but that is not a group itself. Right now this set is composed by Table and Array objects. In fact, Table and Array classes inherit functionality from this class using the *mix-in* technique.

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

4.6 The Table class

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

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

4.6.1 Table instance variables

name The node name.

title The title for this node.

record The record object for this table.

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

varnames The field names for the table.

varnames The typecodes for the table fields.

4.6.2 Table methods

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

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

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

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

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

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

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

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

flush() Flush the table buffers.

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

4.7 The Array class

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

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

4.7.1 Array instance variables

name The node name.

title The node title.

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

typecode The typecode of the represented array.

4.7.2 Array methods

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

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

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

close() Close the array on file.

Appendix A

PyTables Supported Data Types

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

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

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

Type Code	Description	C Type	Size (in bytes)	Python Counterpart
<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/
4. *The HDF5 table programming model*. Examples on using HDF5 tables with the C API. http://hdf.ncsa.uiuc.edu/HDF5/hdf5_hl/doc/RM_hdf5tb_ex.html
5. *HL-HDF*. A High Level Interface to the HDF5 File Format. <ftp://ftp.ncsa.uiuc.edu/HDF/HDF5/contrib/hl-hdf5/README.html>
6. *On the 'Pythonic' treatment of XML documents as objects(II)*. Article describing XML Objectify, a Python module that allows working with XML documents as Python objects. Some of the ideas presented here are used in PyTables. <http://www-106.ibm.com/developerworks/xml/library/xml-matters2/index.html>
7. *gnosis.xml.objectify*. This module is part of the Gnosis utilities, and allows to create a mapping between any XML element to "native" Python objects. http://gnosis.cx/download/Gnosis_Utils-current.tar.gz
8. *Pyrex*. A Language for Writing Python Extension Modules. <http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex>
9. *NetCDF (network Common Data Form)*. This is an interface for array-oriented data access and a library that provides an implementation of the interface. <http://www.unidata.ucar.edu/packages/netcdf/>
10. *NetCDF module on Scientific Python*. ScientificPython is a collection of Python modules that are useful for scientific computing. Its NetCDF module is a powerful interface for NetCDF data format. http://starship.python.net/~hinsen/ScientificPython/ScientificPythonManual/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/>