# Numerical Computing with Python, Lecture 6: Data and I/O

Ramses van Zon

November 28, 2019

## Basic File Input and Output in Python

- Files contain your data
- Files are organized in directories or folders
- A directory is a file too
- Path: sequence of directories to get to a file

## Basic File Input and Output in Python

- Files contain your data
- Files are organized in directories or folders
- A directory is a file too
- Path: sequence of directories to get to a file

**Python modules/packages for files:**

- Built-in Python file objects
- `os` (particularly `os.path`)
- `shutil`
- `pickle`, `shelve`
- `json`, `zipfile`, `tarfile`, `csv`, ...
- `numpy`, `scipy.io.netcdf`, `pytables`, `pandas`, ...

# Basic (Text) File Input and Output in Python

- Get current directory:
  `os.getcwd()`
- Create directory:
  `os.mkdir('folder1')`
- Change current directory:
  `os.chdir('folder1')`
  `os.chdir('..')`
- Get file list:
  `os.listdir('folder1')`
- Get file list by wildcard pattern:
  `glob.glob('folder1/*')`
- Path manipulations: `os.path` module.

# Basic (Text) File Input and Output in Python

- Get current directory:
  `os.getcwd()`
- Create directory:
  `os.mkdir('folder1')`
- Change current directory:
  `os.chdir('folder1')`
  `os.chdir('..')`
- Get file list:
  `os.listdir('folder1')`
- Get file list by wildcard pattern:
  `glob.glob('folder1/*')`
- Path manipulations: `os.path` module.

- Open file for read,write,read/write,append:
  ```
  f = open('folder1/world.txt','r')
  f = open('folder1/world.txt','w')
  f = open('folder1/world.txt','r+')
  f = open('folder1/world.txt','a')
  ```
- Write to file:
  ```
  print(....,file=f)
  f.write("Some line of text\n")
  ```
- Read file:
  ```
  f.read()
  f.readlines()
  ```
- Read line by line: `f.readline()`
- Get/set file pointer:
  `f.tell(), f.seek(position)`
- Close file: `f.close()`

## Metadata

File metadata describes the file and its properties:

- File name
- File size
- Location on disk
- File type (often through magic identifiers)
- Dates/times
- Read/write permissions
- ...

## Metadata

File metadata describes the file and its properties:

- File name
- File size
- Location on disk
- File type (often through magic identifiers)
- Dates/times
- Read/write permissions
- . . .

**Getting the file metadata in Python**

```
import os, stat
from datetime import datetime
```

*Size:*

```
>>> os.path.getsize('folder1/world.txt')
18
```

*Permissions (linux)*

```
>>> st=os.stat('folder1/world.txt')
>>> st.st_mode & stat.S_IXUSR
0
>>> st.st_mode & stat.S_IWUSR
128
```

*Change, modification, access time*

```
>>> st.st_ctime, st.st_mtime, st.st_atime
(1574952256.4632373, 1574952256.4632373, 1574952233.
>>> datetime.fromtimestamp(round(st.st_mtime))
datetime(2019, 11, 28, 9, 44, 16)
```

## Content metadata

File content metadata described properties of the data stored in the file:

- What is the data (*labeled data*)?
- Where did it come from?
- Who made it/owns it/....?
- Format of the data
- Units

This metadata is **not** kept by the file system. One could use a separate metadata file for this.

## Content metadata

File content metadata described properties of the data stored in the file:

- What is the data (*labeled data*)?
- Where did it come from?
- Who made it/owns it/. . . .?
- Format of the data
- Units

This metadata is **not** kept by the file system. One could use a separate metadata file for this.

Formats like netCDF and HDF5, which have interfaces for many different programming languages, allow you to add metadata in the data file itself, and allow to add descriptions.

## Content metadata

File content metadata described properties of the data stored in the file:

- What is the data (*labeled data*)?
- Where did it come from?
- Who made it/owns it/. . . .?
- Format of the data
- Units

This metadata is **not** kept by the file system. One could use a separate metadata file for this.

Formats like netCDF and HDF5, which have interfaces for many different programming languages, allow you to add metadata in the data file itself, and allow to add descriptions.

Even before the data is saved in the file, you need to have tracked of all of this metadata.

Python objects know their data format in memory, and this can be uses by Python packages to write out data including some metadata like the format.

Furthermore, the `xarray` and `pandas` packages can be very convenient in dealing with labeled data.

## I/O Tips

**Minimize I/O Operations**

- Disk **I/O** is usually the **slowest** part of a pipe line.
- When manipulating data from files, try and minimize I/O operations per seconds (IOPS).

# I/O Tips

## Minimize I/O Operations

- Disk **I/O** is usually the **slowest** part of a pipe line.
- When manipulating data from files, try and minimize I/O operations per seconds (IOPS).

```
>>> s = 'Hi world\n'
>>> for c in s:
...     f = open('hiworld.txt','a')
...     f.write(c)
...     f.close()
```

## I/O Tips

### Minimize I/O Operations

- Disk I/O is usually the **slowest** part of a pipe line.
- When manipulating data from files, try and minimize I/O operations per seconds (IOPS).

```
>>> s = 'Hi world\n'
>>> for c in s:
...     f = open('hiworld.txt','a')
...     f.write(c)
...     f.close()
```

```
>>> s = 'Hi world\n'
>>> f = open('hiworld.txt','w')
>>> f.write(s)
>>> f.close()
>>>
```

# I/O Tips, continued

## Close files automatically

- Close a file when done, to flush any buffers and ensures that what is written actually gets to disk.
- But it's easy to forget.
- The `with` statement can automatically close the file for you:

```
>>> with open('hiworld.txt','w') as f:
...     f.write('Hi world!\n')
...
```

## I/O Tips, continued

**Be nice to the file system**

- Don't create millions of files: each requires metadata activity, leading to unwieldy and slow-to-access directories.
- Stick to letters, numbers, underscores and periods in file names.
- Try to reuse data or do more in memory.
- Do not use the file system as a means of communication.

## I/O Tips, continued

**Write numerical data in binary**

- If your data is not text, do not save it as text.
- Let's expand on this one a bit.

# What's in a file?

## Text

- Seems attractive: you can just read it.
- This is not as trivial as it may sound.
- Must assign a bit pattern to each letter or symbol (encoding).
- Ideally unique assignment across languages.

## Binary

- Output the numbers as they are stored in memory
- Avoids expensive conversion to text:
  Writing 128M doubles as text: 173 sec
  Writing 128M doubles in binary: 6 sec
- Not human readable, but is that really so bad?
  If you have 100 million numbers in a file, is any human ever going to read them all?

## Binary Formats

You could invent your own binary format, but it's better to take an existing standard: Saves you potential bugs, the burden of documentation and/or maintaining an IO library, as one probably already exists.

**numpy:** Has a binary format called `npy` or `npz`.

**netCDF:** A self-describing format: contains not only data but names, descriptions of multi-dimensional arrays

Packages: `scipy.io.netcdf` and `netCDF4`

**HDF5:** Another standard, self-describing format
Package: `PyTables` (but called `tables` inside python)
Like a filesystem with annotations, all in one file.

For both netCDF and HDF5, there are tools to inspect/analyze the files. Won't discuss HDF5 here.

# Pickle

Pickle is a Python-native way to store (almost) any data object

- Base64 encoding using "readable" ASCII
- Portable for the same version of Python.
- In the `pickle` module.
- Flexible, can `serialize` any structure.

```
>>> import pickle, os
>>> import numpy as np
>>> a = np.zeros((10000,10000))
>>> f = open('a.pickle','w')
>>> pickle.dump(a,f)
>>> close(f)
>>> print(os.path.getsize('a.pickle'))
3200000198
>>> g = open('a.pickle','r')
>>> b = pickle.load(g)
>>> g.close()
```

pickle.dump wall time: 121.44 s

## NumPy I/O Routines for dealing with arrays in files

- The numpy file formats remember shape of the arrays
- Straight binary dump of data
- Surprisingly simple format but not ported to other languages much.
- Just for NumPy arrays

```
>>> import os
>>> import numpy as np
>>> a = np.zeros((10000,10000))
>>> np.save('a.npy', a)
>>> print(os.path.getsize('a.npy'))
799997952
>>> b = np.load('a.npy')
```

numpy.save wall time: 1.21 s

## Numpy I/O Routines

`save(FILE,ARRAY)`

- save a NumPy array to a .npy file

`savez(FILE, NAME1=ARRAY1, NAME2=ARRAY2)`

- save several NumPy arrays to an uncompressed zipped file with extension .npz

`savez_compressed(FILE, NAME1=ARRAY1, NAME2=ARRAY2)`

- save several NumPy arrays to a compressed zipped file with extension .npz

`load(FILE)`

- load NumPy array(s) from .npy (.npz) file. If FILE is an .npz, a dictionary with keys equal to the names supplied to savez is returned.

## netCDF and HDF

Both are standardized file formats for scientific data, which are:

- Self-describing
- Binary format
- Many tools use these formats
- Parallel access (netCDF4 and HDF5)

## netCDF

- A format as well as an Applications Program interface (API).
- netCDF gives you a higher level approach to writing and reading multi-dimensional arrays.
- Two main versions: netCDF-3 and netCDF-4.

## netCDF-3 files

There are three sections to a netCDF-3 file

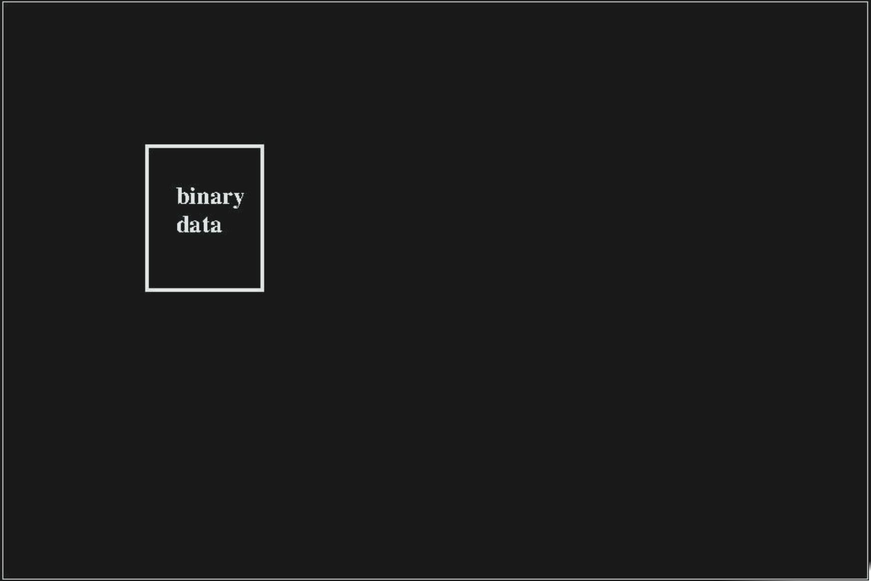
| | |
|---|---|
| **Dimensions** | How many points in each direction of our multidimensional array? |
| **Variables** | The data in our multidimensional array |
| **Attributes** | Variable and other annotations (e.g. units) |

**Python modules**

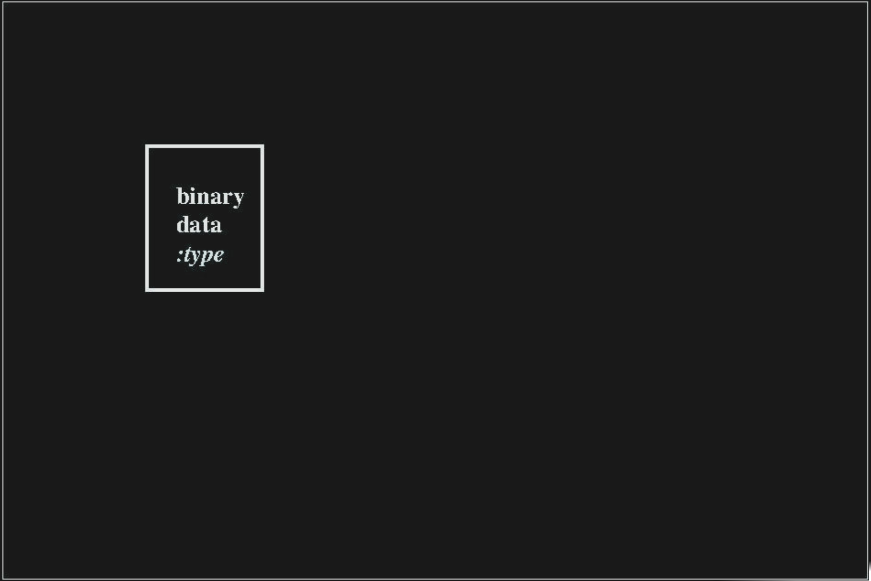
- scipy.io.netcdf: for netcdf3 files
- netCDF4: for netcdf4 files

## netCDF-3 Data Model

## netCDF-3 Data Model



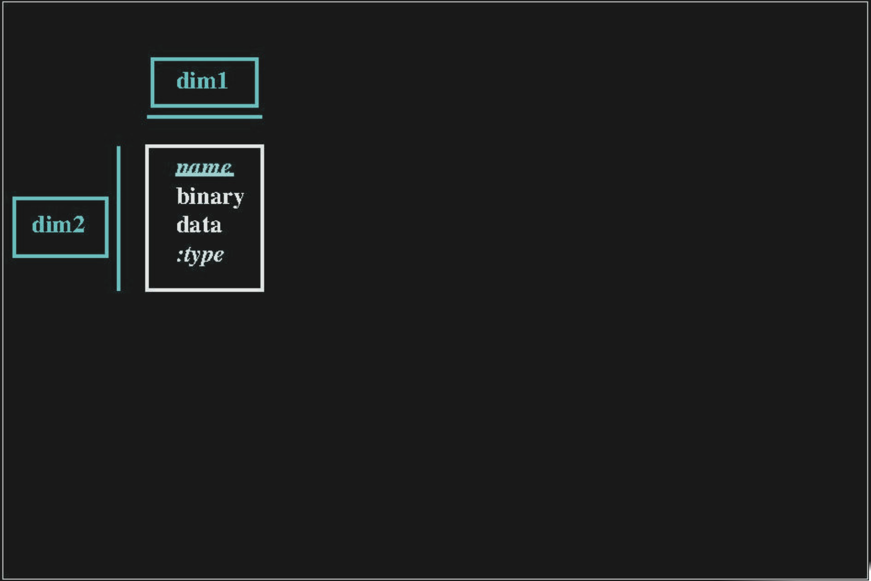
## netCDF-3 Data Model



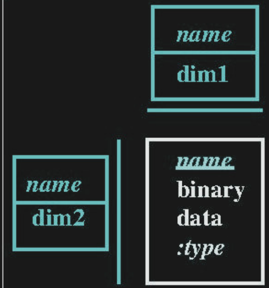
## netCDF-3 Data Model

name
binary
data
:type

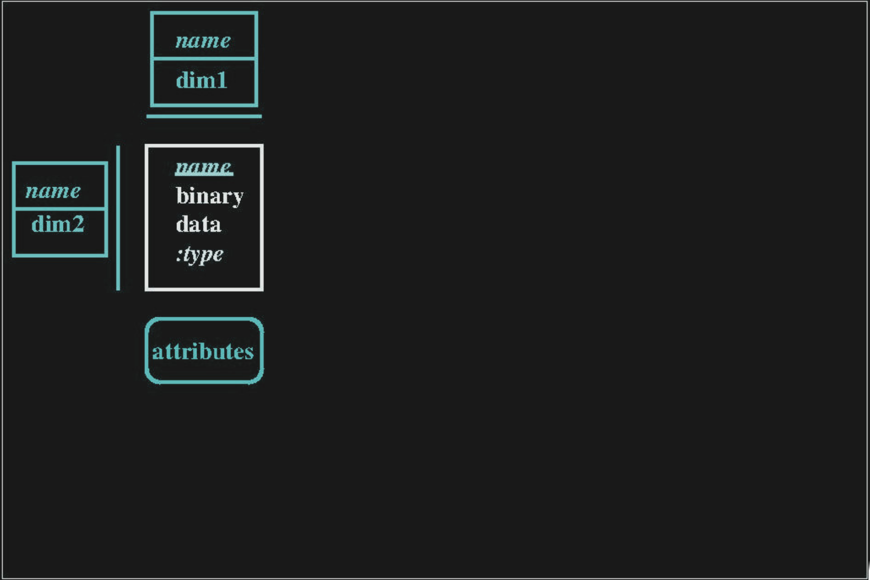
## netCDF-3 Data Model



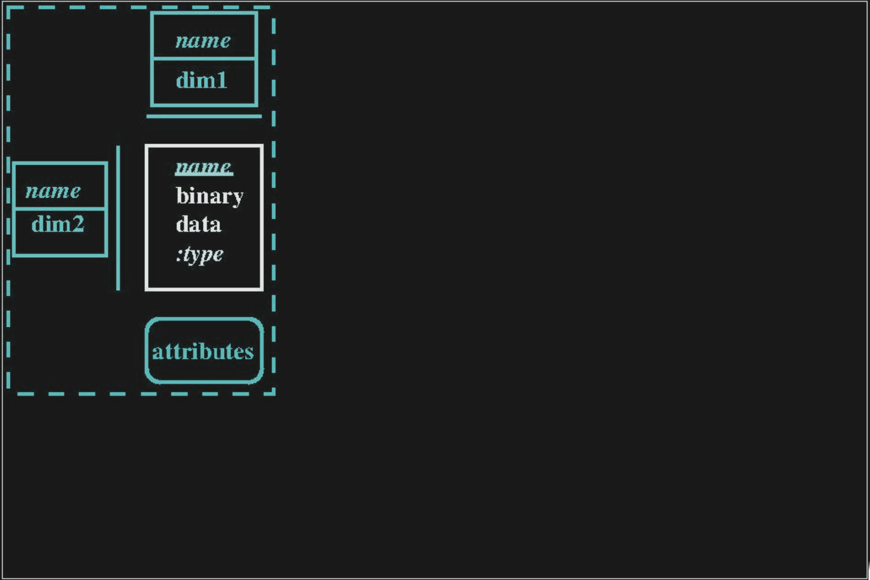
# netCDF-3 Data Model



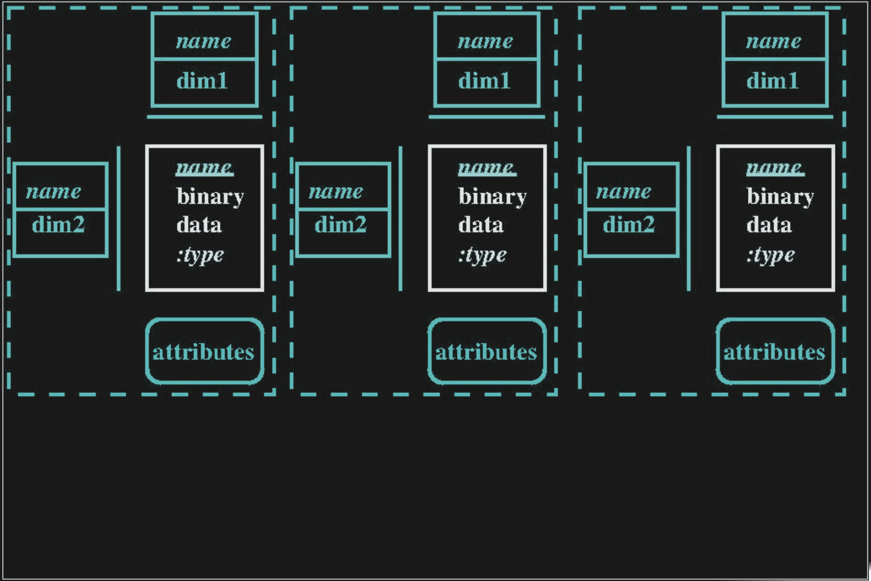
# netCDF-3 Data Model



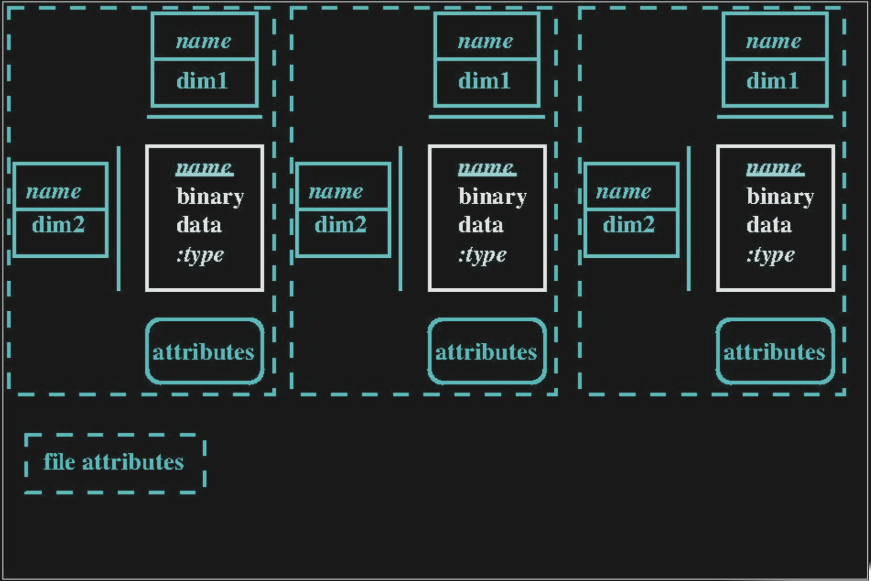
## netCDF-3 Data Model



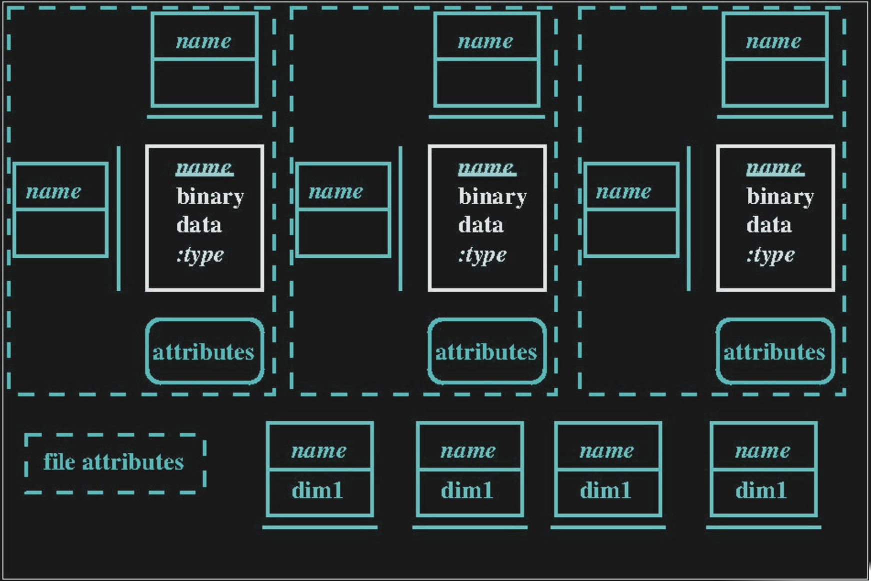
## netCDF-3 Data Model



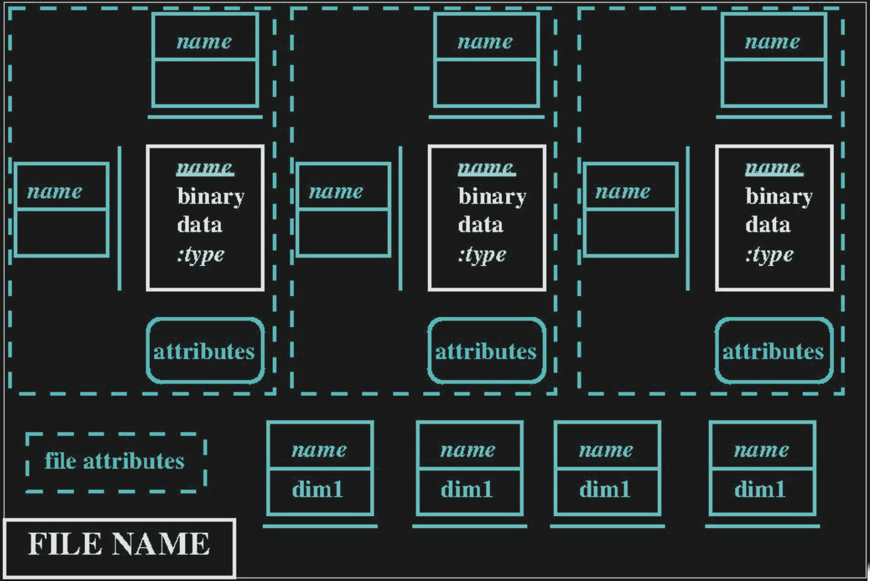
# netCDF-3 Data Model



# netCDF-3 Data Model



# netCDF-3 Data Model



## netCDF example

Can check the 'header' of an netcdf file using the linux utility `ncdump`:

```
$ ncdump -h test.nc
netcdf test {
dimensions:
    x = 1000 ;
variables:
    double a(x, x) ;
        a:units = "Kelvin" ;

// global attributes:
        :history = "This is a test" ;
}
```

Let's see how to create and use this file with the `scipy.io.netcdf` API.

## scipy.io.netcdf: read and write files

### Writing

```
>>> from scipy.io.netcdf import *

>>> f = netcdf_file('test.nc','w')    #create file

>>> f.history = 'This is a test'  # file attribute

>>> f.createDimension('x', 1000) #create dimension

>>> a = f.createVariable('a','d',('x','x')) #array
>>> a[:] = np.zeros((1000,1000))             #fill

>>> a.units = 'Kelvin'               #array attribute

>>> f.close()                 #close file. Important!
```

## scipy.io.netcdf: read and write files

### Writing

```
>>> from scipy.io.netcdf import *

>>> f = netcdf_file('test.nc','w')     #create file

>>> f.history = 'This is a test'  # file attribute

>>> f.createDimension('x', 1000) #create dimension

>>> a = f.createVariable('a','d',('x','x')) #array
>>> a[:] = np.zeros((1000,1000))              #fill

>>> a.units = 'Kelvin'                #array attribute

>>> f.close()                  #close file. Important!
```

### Reading

```
>>> from scipy.io.netcdf import *

>>> f = netcdf_file('test.nc','r')

>>> print(f.history)
This is a test

>>> a = f.variables['a']

>>> print(a[100,300], a.units)
0.0 Kelvin

>>> f.close()
```

# scipy.io.netcdf overview

`HANDLE = netcdf_file(FILENAME, MODE)`

- Opens a netcdf file. `MODE='w'` for writing, `'r'` for reading, `MODE='rw'` for both.

`HANDLE.ATTRIBUTE = VALUE`

- Sets a file `ATTRIBUTE` to the value `VALUE`

`HANDLE.createDimension(NAME, VALUE)`

- Sets the dimension `NAME` (a string) to `VALUE`

`HANDLE.createVariable(NAME, SHAPE)`

- Creates the variable `NAME` with `SHAPE` (tuple of strings as assigned with `createDimension`)

`HANDLE.variables[NAME]`

- The array variable `NAME`

`HANDLE.variables[NAME].ATTRIBUTE = VALUE`

- Set an attribute `ATTIBUTE` of the array variable `NAME` to the value `VALUE`

`HANDLE.close()`

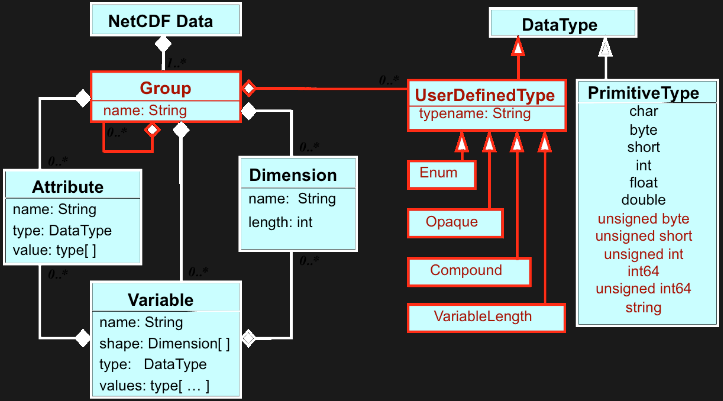
- Flush everything to disk and close the file.

# netCDF-4

Adds some features:

- 64-bit offsets (allowing larger files)
- User defined data types
- Groups
- Parallel I/O
- Built upon HDF-5,
  an even more general format

Python package: `netCDF4`



For simple cases, the netCDF4 API is largely the same as scipy.io.netcdf, except that you open files with `netCDF4.Dataset(...)`.

Nice intro: http://pyhogs.github.io/intro_netcdf4.html

## Labeled data in Python

There are a number of packages that enable working with labeled data, i.e., data with metadata attached.

**pandas**

- Pandas' main functionality centers around labeled tables, i.e., `DataFrames`.
- Tables are composed or row and columns, and those rows and columns can have lables (a bit like `dicts`).
- Each column is saved as a 1d numpy array.
- Pandas offer ways to read and write many different formats, including HDF5.

**xarray**

- Inspired both by the netCDF multi-dimensional array model and pandas.
- Particularly well-suited for multidimensional arrays (which are clumsy to deal with in pandas).
- Offers ways to read and write netCDF files.

## Pandas

- Extremely popular for data science.
- If your data fits in a labeled tabular format, use pandas!
- Functionality centers around the `DataFrame`, which is a labeled table of rows and columns.
- The DataFrame can be viewed as an efficient extension of the Python dictionary.

  A dict is a key-value store.

  A DataFrame is a a key-value-value-... store, where the keys are the labels of the individual rows.
- The separate columns are also labeled.
- The data type of the values in a column must all be the same (and are stored as numpy arrays).

## Pandas example

```
>>> import pandas as pd
>>> name = ['Anna','William','Emma','John','James','Mary']
>>> gender = ['F', 'M', 'F', 'M', 'M', 'F']
>>> number = [2604, 9532, 2003, 9655, 5927, 7065]
>>> data = list(zip(name, gender, number))
>>> print(data)
[('Anna', 'F', 2604), ('William', 'M', 9532), ('Emma', 'F', 2003), ('John', 'M', 9655),
('James', 'M', 5927), ('Mary', 'F', 7065)]
>>> births = pd.DataFrame(data,columns=['Name','Gender','Number'])
>>> births.to_hdf('births1880.h5',key='births')
>>> print(births)
```

| | Name | Gender | Number |
|---|---|---|---|
| 0 | Anna | F | 2604 |
| 1 | William | M | 9532 |
| 2 | Emma | F | 2003 |
| 3 | John | M | 9655 |
| 4 | James | M | 5927 |
| 5 | Mary | F | 7065 |

## xarray

- A newer packages for having labeled multi-dimensional arrays.
- That means your "axes" of the multidimensional arrays could have labels.
- xarray can read and write netcdf files

```
>>> import xarray as xr
>>> ds = xr.open_dataset("test.nc")
>>> print(ds)
<xarray.Dataset>
Dimensions:  (x: 1000)
Dimensions without coordinates: x
Data variables:
    a        (x, x) float64 ...
Attributes:
    history:  This is a test
>>> ds['a'][10,20]
array(0.)
>>> float(ds['a'][10,20])
```

Read more at http://xarray.pydata.org/en/stable