03-Creating_NetCDF

Stephen Pascoe

March 16, 2014

1 Creating NetCDF data in Python

This notebook is based on the Tutorial for the `netcdf4-python` module documented at http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF4-module.html

1.1 NetCDF Model Revision

When working with NetCDF files it is useful to bear in mind the differences between NetCDF3 and NetCDF4 as some tools support both whereas others are not aware of NetCDF4. A full description of these differences is outside the scope of this module. However, these two diagrams from Unidata provide an excellent summary.

The NetCDF Classic Model

The NetCDF Extended Model

1.2 Creating/Opening/Closing a netCDF file

To create a netCDF file from python, you simply call the `Dataset()` constructor. This is also the method used to open an existing netCDF file. If the file is open for write access (`w, r+` or `a`), you may write any type of data including new dimensions, groups, variables and attributes.

NetCDF files come in several flavors (`NETCDF3_CLASSIC, NETCDF3_64BIT, NETCDF4_CLASSIC`, and `NETCDF4`). The first two flavors are supported by version 3 of the netCDF library. `NETCDF4_CLASSIC` files use the version 4 disk format (HDF5), but do not use any features not found in the version 3 API. They can be read by netCDF 3 clients only if they have been relinked against the netCDF 4 library. They can also be read by HDF5 clients. `NETCDF4` files use the version 4 disk format (HDF5) and use the new features of the version 4 API.

The `netCDF4` module can read and write files in any of these formats. When creating a new file, the format may be specified using the `format` keyword in the `Dataset` constructor. The default format is `NETCDF4`. To see how a given file is formatted, you can examine the `file_format` `Dataset` attribute. Closing the netCDF file is accomplished via the `Dataset.close()` method.

This tutorial will focus exclusively on the NetCDF-classic data model. Depending on how the file is created NetCDF-classic files might be HDF5 format on disk or one of the version 3 file formats. From inside Python both formats look the same except for their `file_format` attribute.

Here's an example:

In [1]:
```
from netCDF4 import Dataset
# Create HDF5 *format*, classic *model*
dataset = Dataset('data/test.nc', 'w', format='NETCDF4_CLASSIC')
print dataset.file_format
```

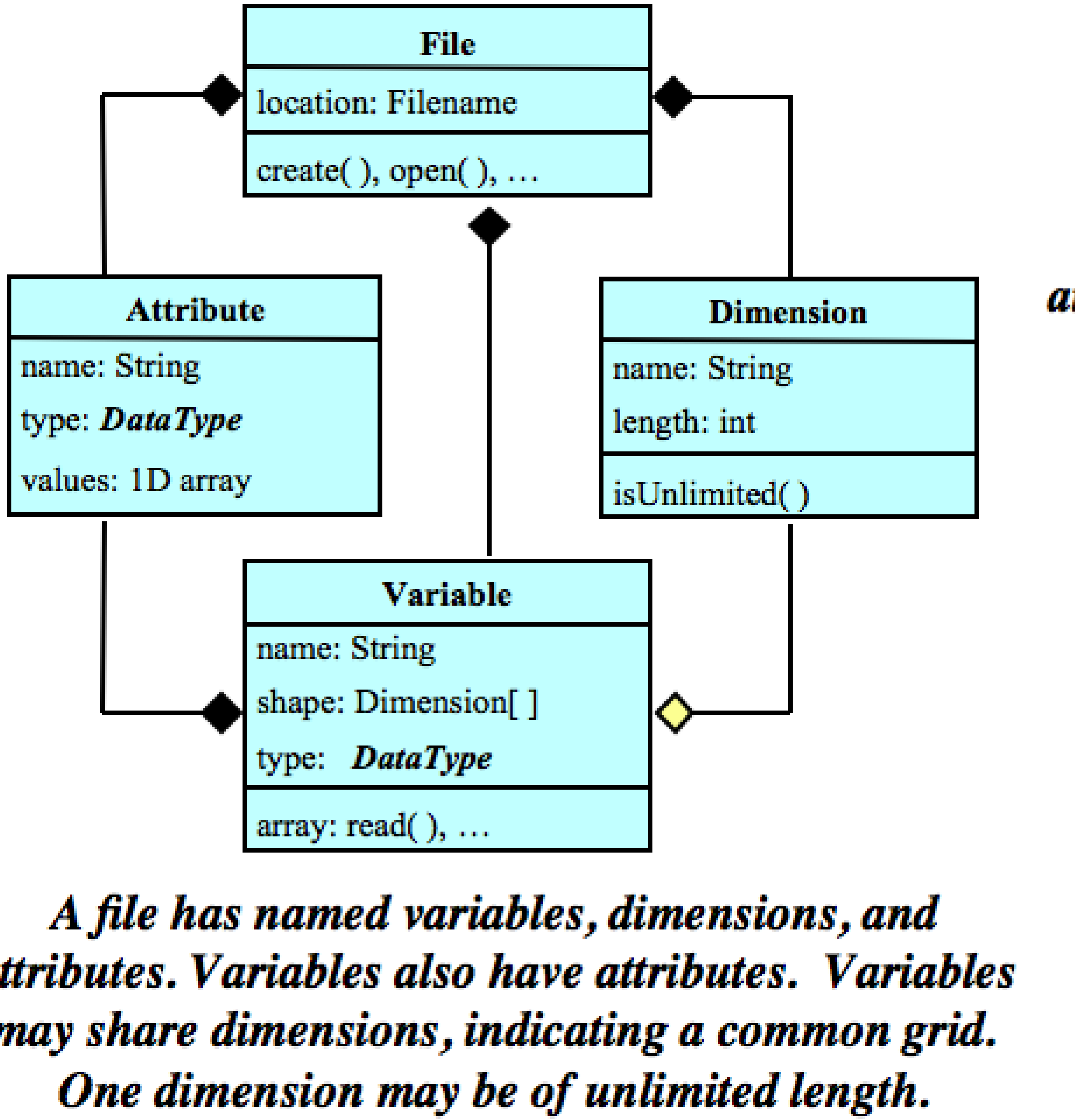


Figure 1: Classic Model

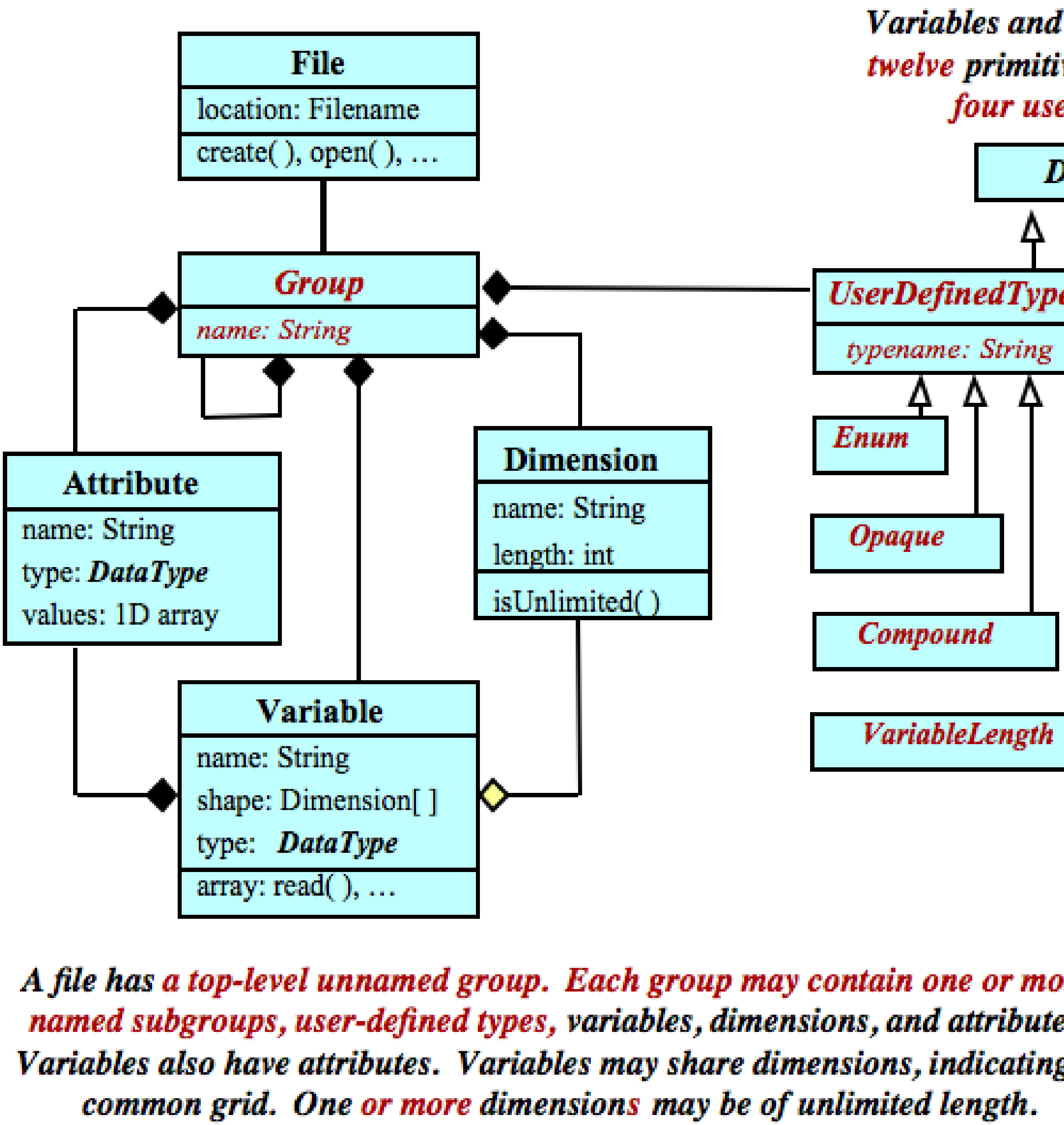


Figure 2: Extended Model

```
NETCDF4_CLASSIC
```

1.3 Dimensions in a netCDF file

netCDF defines the sizes of all variables in terms of dimensions, so before any variables can be created the dimensions they use must be created first. A special case, not often used in practice, is that of a scalar variable, which has no dimensions. A dimension is created using the `Dataset.createDimension` method of a `Dataset`.

A Python string is used to set the name of the dimension, and an integer value is used to set the size. To create an unlimited dimension (a dimension that can be appended to), the size value is set to `None` or `0`. In this example, the `time` dimension is unlimited. In netCDF4 files you can have more than one unlimited dimension, in netCDF 3 files there may be only one, and it must be the first (leftmost) dimension of the variable.

In [2]:
```
level  = dataset.createDimension('level', 10)
time   = dataset.createDimension('time', None)
lat    = dataset.createDimension('lat', 73)
lon    = dataset.createDimension('lon', 144)
```

Calling the python `len()` function with a `Dimension` instance returns the current size of that dimension. The `Dimension.isunlimited()` method can be used to determine if the dimensions is unlimited, or appendable.

In [3]:
```
print len(lon)
```

```
144
```

In [4]:
```
print lon.isunlimited()
```

```
False
```

In [5]:
```
print time.isunlimited()
```

```
True
```

All of the `Dimension` instances are stored in a python dictionary. This allows you to access each dimension by its name using dictionary key access

In [6]:
```
print 'Lon dimension:', dataset.dimensions['lon']

for dimname in dataset.dimensions.keys():
    dim = dataset.dimensions[dimname]
    print dimname, len(dim), dim.isunlimited()
```

```
Lon dimension: <type 'netCDF4.Dimension'>: name = 'lon', size = 144

level 10 False
time 0 True
lat 73 False
lon 144 False
```

1.4 Variables in a netCDF file

netCDF variables behave much like python multidimensional array objects supplied by the `numpy` module. However, unlike numpy arrays, netCDF4 variables can be appended to along one or more 'unlimited' dimensions. To create a netCDF variable, use the `Dataset.createVariable()` method. The `Dataset.createVariable()` method has two mandatory arguments, the variable name (a Python string), and the variable datatype. The variable's dimensions are given by a tuple containing the dimension names (defined previously with `Dataset.createDimension()`).

To create a scalar variable, simply leave out the dimensions keyword. The variable primitive datatypes correspond to the dtype attribute of a numpy array. You can specify the datatype as a numpy dtype object, or anything that can be converted to a numpy dtype object. The unsigned integer types and the 64-bit integer type can only be used if the file format is `NETCDF4`.

The dimensions themselves are usually also defined as variables, called coordinate variables. The `Dataset.createVariable()` method returns an instance of the `Variable` class whose methods can be used later to access and set variable data and attributes.

In [7]:
```
import numpy as np

times      = dataset.createVariable('time',np.float64, ('time',))
levels     = dataset.createVariable('level',np.int32, ('level',))
latitudes  = dataset.createVariable('latitude',np.float32, ('lat',))
longitudes = dataset.createVariable('longitude',np.float32, ('lon',))

temp = dataset.createVariable('temp',np.float32,('time','level','lat','
```

All of the variables in the `Dataset` are stored in a Python dictionary, in the same way as the dimensions:

In [8]:
```
>>> print 'temp variable:', dataset.variables['temp']

>>> for varname in dataset.variables.keys():
...     var = dataset.variables[varname]
...     print varname, var.dtype, var.dimensions, var.shape
```

```
temp variable: <type 'netCDF4.Variable'>
float32 temp(u'time', u'level', u'lat', u'lon')
unlimited dimensions = (u'time',)
current size = (0, 10, 73, 144)

time float64 (u'time',) (0,)
level int32 (u'level',) (10,)
latitude float32 (u'lat',) (73,)
longitude float32 (u'lon',) (144,)
temp float32 (u'time', u'level', u'lat', u'lon') (0, 10, 73, 144)
```

1.5 Attributes in a netCDF file

There are two types of attributes in a netCDF file, global and variable. Global attributes provide information about the entire dataset, as a whole. `Variable` attributes provide information about one of the variables. Global attributes are set by assigning values to `Dataset` instance variables. `Variable` attributes are set by assigning values to `Variable` instances variables. Attributes can be strings, numbers or sequences. Returning to our example,

In [9]:
```
import time

# Global Attributes
dataset.description = 'bogus example script'
dataset.history = 'Created ' + time.ctime(time.time())
dataset.source = 'netCDF4 python module tutorial'

# Variable Attributes
latitudes.units = 'degrees north'
longitudes.units = 'degrees east'
levels.units = 'hPa'
temp.units = 'K'
times.units = 'hours since 0001-01-01 00:00:00.0'
times.calendar = 'gregorian'
```

In [10]:
```
print dataset.description
print dataset.history
```

```
bogus example script
Created Thu Mar 13 10:06:04 2014
```

1.6 Writing data to and retrieving data from a netCDF variable

Now that you have a netCDF `Variable` instance, how do you put data into it? You can just treat it like an array and assign data to a slice.

In [11]:
```
lats = np.arange(-90,91,2.5)
lons = np.arange(-180,180,2.5)
latitudes[:] = lats
longitudes[:] = lons
print 'latitudes =\n',latitudes[:]
```

```
latitudes =
[-90.  -87.5 -85.  -82.5 -80.  -77.5 -75.  -72.5 -70.  -67.5 -65.
-62.5
 -60.  -57.5 -55.  -52.5 -50.  -47.5 -45.  -42.5 -40.  -37.5 -35.
-32.5
 -30.  -27.5 -25.  -22.5 -20.  -17.5 -15.  -12.5 -10.   -7.5  -5.
-2.5
   0.    2.5   5.    7.5  10.   12.5  15.   17.5  20.   22.5  25.
27.5
  30.   32.5  35.   37.5  40.   42.5  45.   47.5  50.   52.5  55.
57.5
  60.   62.5  65.   67.5  70.   72.5  75.   77.5  80.   82.5  85.
87.5
  90. ]
```

Unlike NumPy's array objects, netCDF `Variable` objects with unlimited dimensions will grow along those dimensions if you assign data outside the currently defined range of indices.

In [12]:
```
print 'temp shape before adding data = ',temp.shape
```

```
temp shape before adding data =  (0, 10, 73, 144)
```

In [13]:
```
from numpy.random import uniform

nlats = len(dataset.dimensions['lat'])
nlons = len(dataset.dimensions['lon'])
temp[0:5,:,:,:] = uniform(size=(5,10,nlats,nlons))

print 'temp shape after adding data = ',temp.shape
```

```
temp shape after adding data =  (5, 10, 73, 144)
```

In [14]:
```
# now, assign data to levels dimension variable.
levels[:] =  [1000.,850.,700.,500.,300.,250.,200.,150.,100.,50.]
```

Note: There are some differences between NumPy and netCDF variable slicing rules. Boolean array and integer sequence indexing behaves differently for netCDF variables than for numpy arrays. See the netCDF4-python documentation for details.Time coordinate values pose a special challenge to netCDF users. Most metadata standards (such as CF and COARDS) specify that time should be measure relative to a fixed date using a certain calendar, with units specified like `hours since YY:MM:DD hh-mm-ss`. These units can be awkward to deal with, without a utility to convert the values to and from calendar dates. The functione called `num2date()` and `date2num()` are provided with this package to do just that. Here's an example of how they can be used:

In [15]:
```
# fill in times.
from datetime import datetime, timedelta
from netCDF4 import num2date, date2num

dates = []
for n in range(temp.shape[0]):
    dates.append(datetime(2001, 3, 1) + n*timedelta(hours=12))
times[:] = date2num(dates,
                    units=times.units,
                    calendar=times.calendar)
print 'time values (in units %s): ' % times.units+'\n',times[:]
```

```
time values (in units hours since 0001-01-01 00:00:00.0):
[ 17533104.  17533116.  17533128.  17533140.  17533152.]
```

In [16]:
```
dates = num2date(times[:],
                units=times.units,
                calendar=times.calendar)
print 'dates corresponding to time values:\n',dates
```

```
dates corresponding to time values:
[2001-03-01 00:00:00 2001-03-01 12:00:00 2001-03-02 00:00:00
 2001-03-02 12:00:00 2001-03-03 00:00:00]
```

`num2date()` converts numeric values of time in the specified `units` and `calendar` to datetime objects, and `date2num()` does the reverse. All the calendars currently defined in the CF metadata convention are supported. A function called `date2index()` is also provided which returns the indices of a netCDF time variable corresponding to a sequence of datetime instances.

In [17]:
```
dataset.close()
```

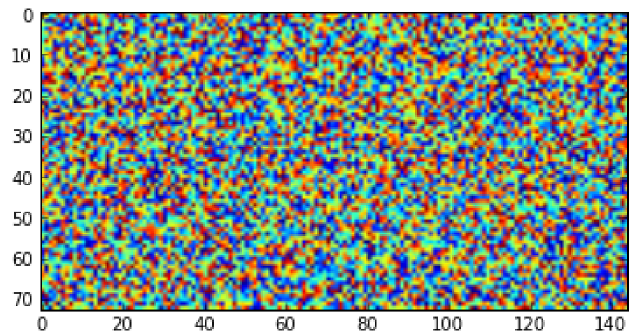
In [18]:
```
%%sh
ncdump -h data/test.nc
```

```
netcdf test {
dimensions:
        level = 10 ;
        time = UNLIMITED ; // (5 currently)
        lat = 73 ;
        lon = 144 ;
variables:
        double time(time) ;
                time:units = "hours since 0001-01-01 00:00:00.0" ;
                time:calendar = "gregorian" ;
        int level(level) ;
                level:units = "hPa" ;
        float latitude(lat) ;
                latitude:units = "degrees north" ;
        float longitude(lon) ;
                longitude:units = "degrees east" ;
        float temp(time, level, lat, lon) ;
                temp:units = "K" ;

// global attributes:
                :description = "bogus example script" ;
                :history = "Created Thu Mar 13 10:06:04 2014" ;
```

```
                                 :source = "netCDF4 python module tutorial" ;
}
```

In [19]:
```python
from matplotlib import pyplot as plt

d = Dataset('data/test.nc')
temp = d.variables['temp']
plt.imshow(temp[1,1])
```

Out [19]: <matplotlib.image.AxesImage at 0x106055fd0>



TODO: CF compliance of the results

Exercise 3 : Creating NetCDF