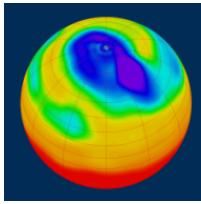
National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL

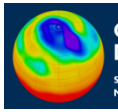


Centre for Environmental Data Analysis
SCIENCE AND TECHNOLOGY FACILITIES COUNCIL
NATURAL ENVIRONMENT RESEARCH COUNCIL

Creating NetCDF files with Python

Stephen Pascoe and Ag Stephens

National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL



Centre for Environmental Data Analysis
SCIENCE AND TECHNOLOGY FACILITIES COUNCIL
NATURAL ENVIRONMENT RESEARCH COUNCIL

We are using netCDF4-python!

There are many options for working with NetCDF files in Python. In this example we have chosen to highlight the use of the **netCDF4-python** module.

The **netCDF4-python** module is useful because:

- It implements the basic "classic" model as well as more advanced features.
- It provides a simple interface to the NetCDF structure.
- It has been used as the underlying NetCDF I/O layer for many more advanced packages.

Creating/Opening 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.

Working with “classic” NetCDF

The `netCDF4` module can read and write files in any netCDF format. When creating a new file, the format may be specified using the `format` keyword in the `Dataset` constructor. The default format is `NETCDF4`.

This tutorial will focus exclusively on the NetCDF-”classic” data model using: `NETCDF4_CLASSIC`

Creating a NetCDF file

Open a new NetCDF file in “write” ('w') mode:

```
>>> from netCDF4 import Dataset
>>> dataset = Dataset('data/test.nc', 'w',
            format='NETCDF4_CLASSIC')

>>> print dataset.file_format
NETCDF4_CLASSIC
```

Create dimensions

Define a set of dimensions used for your variables:

```
level = dataset.createDimension('level', 10)
lat = dataset.createDimension('lat', 73)
lon = dataset.createDimension('lon', 144)
time = dataset.createDimension('time', None)
```

And you can query dimensions with:

```
>>> print len(lon)
144
>>> print time.isunlimited() # is special!
True
```

Dimensions

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:

```
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
```

Variables

NetCDF **variables** behave much like python multi-dimensional arrays in `numpy`. However, unlike numpy arrays, netCDF4 variables can be appended to along the 'unlimited' dimension (a.k.a. the "record dimension").

To create a netCDF variable, use:

```
Dataset.createVariable(<var_id>, <type>,
                       <dimensions>)
```

This method has two mandatory arguments: the variable ID (a Python string) and the variable datatype. Additionally a tuple of dimensions can be provided.

Creating a variable

```
import numpy as np

# Create coordinate variables for 4-dimensions

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',))

# Create the actual 4-d variable
temp = dataset.createVariable('temp', np.float32,
 ('time','level','lat','lon'))
```

Accessing variables

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

```
>>> 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(time, level,
lat, lon) unlimited dimensions: time
current shape = (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)
```

Attributes (global)

Global attributes are set by assigning values to `Dataset` instance variables. Attributes can be strings, numbers or sequences. Returning to our example:

```
import time

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

Attributes (variable)

Variable attributes are set by assigning values to `Variable` instance variables:

```
# Variable Attributes
latitudes.units = 'degree_north'
longitudes.units = 'degree_east'
levels.units = 'hPa'
temp.units = 'K'

times.units = 'hours since 0001-01-01 00:00:00'
times.calendar = 'gregorian'
```

Accessing attributes

Attributes are accessed as attributes of their relevant instances:

```
>>> print dataset.description
bogus example script

>>> print dataset.history
Created Mon Mar 17 01:12:31 2014
```

Writing data

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.

```
>>> 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.
...
-60.    -57.5 -55.     -52.5 -50.     -47.5 -45.     -42.5 -40
...
```

Growing data along unlimited dimension (1)

Unlike NumPy's array objects, netCDF `Variable` objects that have an unlimited dimension will grow along that dimension if you assign data outside the currently defined range of indices.

```
>>> print 'temp shape before adding data = ', temp.shape

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

Growing data along unlimited dimension (2)

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

NOTE: **numpy.random.uniform**(size = X) returns values from a uniform distribution in a numpy array with dimensions expressed in a tuple X.

Defining date/times correctly (1)

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 functions `num2date()` and `date2num()` are provided with this package to do just that. Here's an example of how they can be used…

Defining date/times correctly (2)

```
>>> # 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.]
```

Defining date/times correctly (3)

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

dates corresponding to time values:
  [datetime.datetime(2001, 3, 1, 0, 0)
   datetime.datetime(2001, 3, 1, 12, 0)
   datetime.datetime(2001, 3, 2, 0, 0)
   datetime.datetime(2001, 3, 2, 12, 0)
   datetime.datetime(2001, 3, 3, 0, 0)]
```

`num2date()` converts numeric values of time in the specified `units` and `calendar` to `datetime` objects, and **`date2num()`** does the reverse.

Finally, let's write the file

Simply…

```
>>> dataset.close()
# and the file is written!
```

I wonder what it looks like

```
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 Mon Mar 17 01:12:31 2014" ;
    :source = "netCDF4 python module tutorial" ;
}
```

Further reading

Python-netCDF4:

http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF4-module.html