

Data ingestion and output
(reading and writing data)

Outline

- **ASCII input files**
- **Binary input files**
- **CDMS-compatibility (is the solution to all your I/O problems!)**
- **Writing data to output files**

Some grounding

- Python itself, especially via the **string** module, makes it really easy to manipulate string, and therefore ingest ASCII data.
- The **struct** module, coupled with the **Numeric** package allows for some ingestion of strictly binary files.
- BUT, the biggest strength of CDAT is the addition of the **cdms** module, which allows the user to really easily ingest self-describing binary files such as the ones in the NetCDF format.

Reading text files in Python

- In its simplest form python provides useful tools to read ASCII data via string manipulation:

```
f=open('file.txt')
lines=f.readlines()
for ln in lines:
    sp=ln.split()
    print ln, "splits to:", sp
```

----- Example outputs from above -----

“o3, 0.3462, 0.5834” splits to: [“o3”, “0.3462”, “0.5834”]

“no2, 2.4435, 3.4352” splits to: [“no2”, “2.4435”, “3.4352 ”]

ASCII files via contrib package: asciidata

- The contributed **asciidata** module provides some simple ASCII file reading functions.
- Imagine a file containing tab_delimited data:

var1	var2
22	44.3
34	48.3

```
>>> import asciidata
>>> a=asciidata.tab_delimited('tab_del_data.txt')
>>> print a
{'var1': array([ 22.,  34.]), 'var2': array([ 44.3,
      48.3])}
```

ASCII files via CDAT: using VCDAT's browser module (1)

- The ASCII file reading capabilities of VCDAT can be accessed from the command line via the “browser” module.

- For non-formatted data:

```
browser.gui_ascii.read(text_file ,header=0,  
                        ids=None, shape=None, next='-----  
                        ', separators=[';', ',', ':'])
```

- **header**: number of lines to skip at the beginning of the file
- **ids**: Name(s) to assign to the variables returned
- **shape**: Shape(s) to give to each variable read
- **next**: string separator between each variable
- **separators**: string separating elements

ASCII files via CDAT: using VCDAT's browser module (2)

- For data in columns:

```
browser.gui_ascii_cols.read( text_file ,header=0,  
    cskip=0, cskip_type='columns', axis=0, ids=None,  
    idrow=0, separators=[';',',', ':'])
```

- **cskip**: number of column/character to skip
- **cskip_type**: what to skip column or character
- **axis**: 0/1 is the first column to be used as an axis for the 1D variables
- **idrow**: 0/1 use the first row to set variable ids
- **ids**: name to give to variables returned

ASCII files via contributed package: Scientific (1)

You can read ASCII “Fortran Formatted” files using the Scientific contributed package:

```
>>> f = open(ascii_filename, 'r')
      # Import the module that does the work.
>>> from Scientific.IO import FortranFormat
      # Declare the fortran formats used to create the
      data.
>>> ff1 = FortranFormat.FortranFormat('2i6')
>>> ff2 = FortranFormat.FortranFormat('12i6')
```

ASCII files via contributed package: Scientific (2)

```
>>> data_line = f.readline()
>>> mon, yr = FortranFormat.FortranLine(data_line,
    ff1)# Now define an array to read the data into.
>>> import Numeric
>>> T_array = Numeric.zeros((14,))# In the next line
    you are assigning the values.
>>> T_array[start_index: end_index] =
    FortranFormat.FortranLine(f.readline(), ff2)
# Note: You must have previously defined T_array.
# See tutorial examples for more details.
```

Reading Binary files

- Fortran code also produce “pure” binary file, for this the ***struct*** module can be really useful
- See <http://docs.python.org/lib/module-struct.html> for more details.
- Alternatively you can use the function from VCDAT inside the “browser” module:

```
browser.gui_read_Struct.read( file
    ,format="", endian='@', datatype='f',
    ids=[], shape=[], separator=""):
```

Reading Binary files

- Or you can use the contributed 'binaryio' package:

```
from binaryio import *
iunit = bincreate('filename')
binwrite(iunit, some_array) # (up to 4 dimensions,
    or scalars)
binclose(iunit)
iunit = binopen('filename')
y = binread(iunit, n, ...) # (1-4 dimensions)
binclose(iunit)
```

Self-Describing Binary Files (1)

But the best way to read data in CDAT is to use the **cdms** module. Recognised formats are:

- **NetCDF** (standard for input and output) – CDMS follows the Climate and Forecasts (CF) Metadata Convention for NetCDF.
- **HDF4** – currently incompatible with the NetCDF option due to library conflicts. CDAT can be built with either, not both. There is a hope ahead with a merger planned of NetCDF4 and HDF5 libraries (<http://my.unidata.ucar.edu/content/software/netcdf/netcdf-4/index.html>).

Self-Describing Binary Files (2)

- More recognised format are:
 - **GRIB** – is handled via the GrADS/GRIB interface, a slightly convoluted but effective way to get data into CDAT.
 - **PCMDI DRS** format – not covered here as relatively little UK usage.
 - **CDML** (Climate Data Markup Language) – the internal CDAT XML representation that points to multiple binary files.

Reading GRIB 1

To read GRIB (regular grids only), use the “***grib2ctl.pl***” perl script to generate the control file (“***.ctl***”).

```
dset ^test.grb
index ^test.grb.idx
undef 9.999E+20
title test.grb
* produced by grib2ctl v0.9.12.5p32l
dtype grib 255
options yrev
ydef 181 linear -90.000000 1
xdef 360 linear 0.000000 1.000000
tdef 1 linear 18Z01jan1996 6hr
zdef 21 levels
21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
vars 1
O3hbl 60 203,109,0 ** Ozone mass mixing ratio kg kg**-1
```

**Example
Control
(* .ctl) file**

**[produced by
grib2ctl.pl]**

***grib2ctl.pl* is available at:**

<http://www.cpc.ncep.noaa.gov/products/wesley/grib2ctl.html>

Reading GRIB 2

The 'gribmap' utility (part of GrADS) is used to create a small index file that points to the correct sections of the GRIB file to access the actual data.

Typical usage:

```
$ grib2ctl.pl afile.grb > afile.ctl
```

```
$ gribmap -e -i afile.ctl
```

```
# Open via the "afile.ctl" file.
```

***gribmap* is available as part of GrADS at:**

<http://grads.iges.org/grads/>

Other self-describing formats of interest in the UK

- You can also get support for:
 - **PP-format** – the BADC has developed code for reading the Met Office proprietary field data format. This should soon be included in the I/O layer beneath CDMS (known as cdunif – a C-layer that provides read access to multiple formats, and write access to NetCDF). Ask for details.
 - **NASA Ames** – a group of ASCII formats developed at NASA for field experiments and data exchange. Used extensively in UK atmospheric research. The BADC has developed a NASA Ames I/O Python package that links to cdms (see: <http://home.badc.rl.ac.uk/astephens/software/nappy>).

CDMS (The heart of CDAT!)

CDMS is the python package at the core of CDAT.

You'll spend hours of your life typing "import cdms" when manipulating variables, files and datasets. It provides **the best way to read and write data**:

- Opening a file for reading:

```
f=cdms.open(file_name)
```

- will open an existing file protected against writing

- Opening a new file for writing:

```
f=cdms.open(file_name, 'w')
```

- will create a new file even if it already exists

- Opening an existing file for writing:

```
f=cdms.open(file_name, 'r+') # or 'a'
```

- will open an existing file ready for writing or reading

Basic (NetCDF) File I/O example

- File I/O to NetCDF is simple and dealt with at the object level:

```
import cdms
ufile = cdms.open('u-wind.nc')
vfile = cdms.open('v-wind.nc')

(u, v) = (ufile('u'), vfile('v'))

wind_speed = (u**2 + v**2)**0.5
outfile = cdms.open('wspd.nc', 'w')
outfile.write(wind_speed)
outfile.close()
```

← **cdms.open** function binds **ufile** to an instance of **CdmsFile**

← **u** and **v** are instances of the **FileVariable** class.

← **wind_speed** is a new **TransientVariable** instance

← **outfile** is another **CdmsFile** instance with write permission

Pulling data from a CDMS file object

- Multiple way to retrieve data

- All of it, omitted dimensions are retrieved entirely

```
s=f('var')
```

- Specifying dimension type and values

```
s=f('lnsp', time=(time1,time2))
```

```
# Known types: time, level, latitude, longitude  
(t,z,y,x) # lat=latitude, lon=longitude.
```

- Dimension names and values

```
s=f('temp', <dimname1>=(val1,val2)) # <dimname>  
is 'time', 'level', 'latitude' or 'longitude'.
```

- Sometimes indices are more useful than actual values

```
s=f('tco3', time=slice(index1,index2,step))
```

Axis selection: Time Dimension

The Time dimension provides a special case where you want to specify more than values in the array:

- Raw values are not necessarily meaningful without a reference time.
- 2 Solutions:

- Use strings as “value”

```
s=f(var,time=('2004','2004-4-29 10:30'))
```

- Use **cdtime** object:

```
t1=cdtime.comptime(2004)
```

```
t2=cdtime.comptime(2004,4,29,10,30)
```

```
s=f(var,time=(t1,t2))
```

The Mysterious 3rd argument

OK, we understood $s=f(\text{'var'}, \text{time}=(t1, t2))$, but there is actually a third argument defaulted to 'ccn'. What is it? (i.e. $(t1, t2, \text{'ccn'})$):

- The first 2 letters represents the bounds of the retrieved segment they can be "c" or "o" as in "**C**losed" or "**O**pened":

- » 'cc' : [v1,v2]

- » 'co' : [v1,v2[

- » 'oo' :]v1, v2[

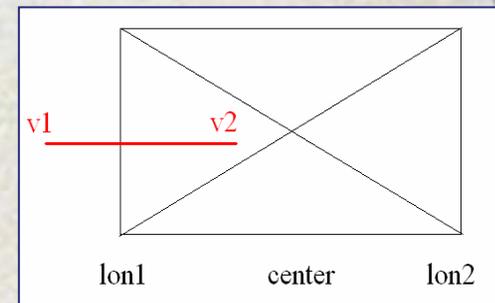
- The third letter represents the search method, it can be 'b', 'n', 'e' or 's' as in '**B**ounds', '**N**ode', '**E**xtranode' or '**S**elect'

- i.e the cell will be considered valid if the bounds or node are within the interval defined In the example figure:

(v1,v2,'ccb') selects

(v1,v2,'ccn') does not select

- 'e': same as n but add an extra node
- 's': select axis elements for which the cell boundary is a subset of the interval



cdms.open() – other options

The `cdms.open()` call also allows other known keywords for data ingestion:

Keyword	Description	Value
<i>axisid</i>	Restrict the axis with ID <i>axisid</i> to a value or range of values.	See Table 2.37 on page 101
<code>grid</code>	Regrid the result to the grid.	Grid object
<code>latitude</code>	Restrict latitude values to a value or range. Short form: <code>lat</code>	See Table 2.37 on page 101
<code>level</code>	Restrict vertical levels to a value or range. Short form: <code>lev</code>	See Table 2.37 on page 101
<code>longitude</code>	Restrict longitude values to a value or range. Short form: <code>lon</code>	See Table 2.37 on page 101
<code>order</code>	Reorder the result.	Order string, e.g., “tzyx”
<code>raw</code>	Return a masked array (MA.array) rather than a transient variable.	0: return a transient variable (default); =1: return a masked array.

cdms.open() – other options

Keyword	Description	Value
required	Require that the axis IDs be present.	List of axis identifiers.
squeeze	Remove singleton dimensions from the result.	0: leave singleton dimensions (default); 1: remove singleton dimensions.
time	Restrict time values to a value or range.	

Writing data to files

- NetCDF – the standard format for geospatial datasets.
- Creating and writing a CDMS variable.
- Writing variables and attributes to a CDMS file.
- Writing non-standard binary files.
- Writing ASCII files.

NetCDF – the standard data format for climate data

- The standard CDAT output format is **NetCDF** (compliant with the **CF-Metadata Convention**)
- Some selling points for choosing NetCDF:
 - NetCDF is used extensively in the atmospheric and oceanic science communities.
 - NetCDF is a portable self-describing binary data format.
 - NetCDF is network-transparent, meaning that it can be accessed by computers that store integers, characters and floating-point numbers in different ways.
 - NetCDF provides direct-access: a small subset of a large dataset may be accessed efficiently, without first reading through all the preceding data.
 - NetCDF is appendable: data can be appended to a NetCDF dataset along one dimension without copying the dataset or redefining its structure.

NetCDF – *more good stuff!*

- NetCDF datasets can be read and written in a number of languages, these include C, C++, FORTRAN, IDL, Python, Perl, and Java.
- The different language implementations are freely available from the UNIDATA ftp area or from other mirror sites.
- Several graphics packages support NetCDF input, making it very easy to display and analyse NetCDF datasets. For instance FERRET and CDAT provide both command line and graphical user interfaces for displaying and analysing gridded data.
- NetCDF is completely and methodically documented in UNIDATA's NetCDF User's Guide.
- Several groups have defined conventions for NetCDF files, to enable the exchange of data. CDAT has adopted the Climate and Forecasting (CF) conventions for NetCDF data.
- *Tell me where to get more information!*

<http://my.unidata.ucar.edu/content/software/netcdf/index.html>

NetCDF – a look inside (1)

- NetCDF tools make it so easy (even on Windows!):

```
$ ncdump [-h] simple.nc # and out comes an  
ASCII listing of the file contents...
```

NetCDF – a look inside (2)

```
netcdf simple {
dimensions:
    latitude = 3 ;
    longitude = 2 ;
    time = UNLIMITED ; // (5 currently);
variables:
    double time(time) ;
        time:standard_name = "time" ;
        time:units = "minutes since 1994-01-01" ;
    float latitude(latitude) ;
        // global attributes:
        :Conventions = "CF-1.0" ;
        :institute = "The British Atmospheric Data Centre." ;
        :source = "Model developed in conjunction with IPLSPSC." ;
        :history = "10 Sep 2002 - Created by hand.\n",
            "18 Mar 2003 - Modified by feet.\n" ;
    float longitude(longitude) ;
        :title = "Model output from imaginary model (temperONETER)." ;
        :comment = "Not very useful data." ;
        :references = "A great report somewhere!" ;
    data:
    float temp(time, latitude, longitude) ;
        temp:time = 0.5, 1.5, 2.5, 3.5, 4.5 ;
        temp:latitude = 54.2, 54.4, 54.6 ;
        temp:longitude = 2.0, 2.5 ;
        temp = 34.5, 31.2, 23.7, 19.6, 35.8, 29.2, 24.4, 5.6, 7.2, 8.1,
            18.6, 15.2, 13.1, 4.6, 3.7, 8.2, 9.7, 34.2, 26.7, 28.7,
            2.1, 3.4, 5.6, 7.8, 9.0, 10.2, 11.2, 11.6, 11.7, 11.8 ;
}
```

Is this relevant?

Yes, because CDAT uses a NetCDF-like structure internally!

Creating a CDMS variable to write to a file

To create a variable from scratch...

- Create a very simple Numeric array:

```
import cdms, Numeric
data_array=Numeric.array([23, 56, 189, 23.4], 'f')
```

- Create a latitude axis:

```
latax=cdms.createAxis([20., 30., 40., 50.])
latax.designateLatitude()
latax.units="degrees_north"
```

- Create a CDMS Transient variable:

```
myvar=cdms.createVariable(data_array, axes=[latax],
    attributes={"long_name": "temperature",
    "units": "K"})
```

You now have a variable that can be written straight to a CDMS file!

Opening a file for writing

- Opening files with CDMS is simple, the second argument dictates what type of access the user has:
 - ‘r’ = read only (default if no second argument provided), existing files only.
 - ‘w’ = write only, creates a new file or overwrites existing.
 - ‘r+’ or ‘a’ = read-write. ‘a’ opens a file if it exists or creates one otherwise.
- To write a new CDMS file:

```
outfile=cdms.open('myfile.nc', 'w')  
# and to close:  
outfile.close()
```
- Same grammar as the built-in open function! This can be a reason to not import everything from CDMS because “**from cdms import ***” will overload the built-in ‘open’ function.

Writing file variables and attributes

- Writing CDMS variables, Numeric arrays or Masked Arrays to a CDMS file object is very easy:

```
outfile.write(myvar)  
outfile.write(another_variable)
```

- Writing file attributes (file level metadata) corresponds to setting global attributes in a NetCDF file and is simply done by setting class attributes:

```
outfile.source="Data from the centre of the  
Galaxy"  
outfile.sauce="Ketchup"  
outfile.version="3.1"
```

Writing non-standard binary files

- Once again you can use the python **struct** module to write more complex binary output files. For simple binary arrays written to files you can use the built-in python I/O:

```
>>> outfile=open("binary_var.dat", "wb")
>>> x=N.array([2,4,6,8,9], 'f')
>>> outfile.write(x)
>>> outfile.close()
```

- If you have a multi-dimensional variable then you might need to grab each row and write them according to your own file format design.
- *But why not use NetCDF?*

Writing ASCII files

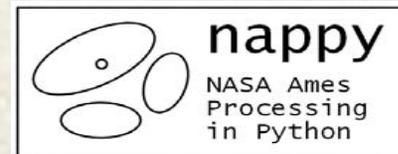
- Writing ASCII files is largely about your preferences as a file author:
 - Do you want any metadata retained?
 - Do you want to follow any standards or make up your own brand?
 - How do you want the data (and/or metadata) formatted?

- The simple view is to open an ASCII file and write to it:

```
>>> outfile=open('my_output.txt', 'w')
>>> outfile.write("Header: CHORDEX34 flight data\n")
>>> outfile.write("Time\tTemp (K)\tWspd (m/s)\n")
>>> times=temp.getTime().asComponentTime()
>>> c=0
>>> while c<len(times):
...     outstring="%s\t%s\t%s\n" % (times[c],
...                                 temp[c], wspd[c])
...     outfile.write(outstring)
...     c=c+1
>>> outfile.close()
```

Writing ASCII files in NASA Ames format (1)

- The BADC has written a package to bridge the gap between the NASA Ames File format(s) developed in the 1990s for data exchange in scientific projects.
- **nappy – NASA Ames Processing in Python** – allows you to write CDMS variables directly to NASA Ames (some sub-formats will choke, but most will work!).



- Get nappy (beta-release) at:

<http://home.badc.rl.ac.uk/astephens/software/nappy>

- Command-line usage:

```
$ cdms2na.py -i cdmsFile.nc -o naFile.na
```

