CDAT Refresher

Author: Charles Doutriaux

February 5th, 2010

Outline

- Python Basics
- Arrays, Masked Arrays, Masked Variables
- Files I/O
- Data analysis
- Visualization
- Mixing with other languages.
- Open Session

Python Basics
Customizing Python

- Significant environment variables:
  - PYTHONPATH : list of additional directories where Python will look for modules to import. It will look there first
    - Example in (t)csh : setenv PYTHONPATH ${HOME}/Python/MyStuff
  - PYTHONSTARTUP : points to a file to execute whenever your start Python by itself. But will not be ran whenever your run a script.
    - Example:
      - setenv PYTHONPATH ${HOME}/.pythonrc
      - python
        - » This will start python, execute the content of the file at ${HOME}/.pythonrc, and then gives you the hand
      - python myscript.py
        - » This will start python execute the content of the file "script.py" but NOT ${HOME}/.pythonrc., and exit
      - python –i myscript.py
        - » As above but gives the hand back to the user for "I"nteractive mode.
    - PYTHNOSTARTUP is useful it can run a few thing that you always need to do, for example import some modules you always need, load some files, etc...

Python Basics
Customizing Python

- Auto-completion.
  - The radline and rlcompleter modules are VERY useful as they help you discover what attributes and methods are available on an object.
  - To obtain these capabilities simply run (or add the following lines to you PYTHONSTARTUP file)
    - import rlcompleter, readline
    - readline.parse_and_bind("tab: complete")
  - Now simply type A. and then hit “tab” to see the possible completions
    - Example: a=1; a. # tab/tab returns
  - The traditional way would be:
    - a=1; print dir(a)

```
a.__abs__(          a.__getattribute__(  a.__new__(          a.__rfloordiv__(    a.__str__(
a.__add__(          a.__getnewargs__(    a.__nonzero__(      a.__rlshift__(      a.__sub__(
a.__and__(          a.__hash__(          a.__oct__(          a.__rmod__(         a.__subclasshook__(
a.__class__(        a.__hex__(           a.__or__(           a.__rmul__(         a.__truediv__(
a.__cmp__(          a.__index__(         a.__pos__(          a.__ror__(          a.__trunc__(
a.__coerce__(       a.__init__(          a.__pow__(          a.__rpow__(         a.__xor__(
a.__delattr__(      a.__int__(           a.__radd__(         a.__rrshift__(      a.conjugate(
a.__div__(          a.__invert__(        a.__rand__(         a.__rshift__(       a.denominator
a.__divmod__(       a.__long__(          a.__rdiv__(         a.__rsub__(         a.imag
a.__doc__           a.__lshift__(        a.__rdivmod__(      a.__rtruediv__(     a.numerator
a.__float__(        a.__mod__(           a.__reduce__(       a.__rxor__(         a.real
a.__floordiv__(     a.__mul__(           a.__reduce_ex__(    a.__setattr__(
a.__format__(       a.__neg__(           a.__repr__(         a.__sizeof__(
```

```
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__',
'__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '__format__',
'__getattribute__', '__getnewargs__', '__hash__', '__hex__', '__index__', '__init__',
'__int__', '__invert__', '__long__', '__lshift__', '__mod__', '__mul__', '__neg__',
'__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'conjugate',
'denominator', 'imag', 'numerator', 'real']
```

Python Basics

- Object oriented, i.e. everything is an "object" which has "methods" (functions) and "attributes" associated with itself.
  - A = 1 ; print A.__add__(2) # returns 3 (which is an object itself)

Python Basics: Important types

- Tuples () and lists [] : They are very similar, except than tuples cannot be altered, they are both ordered
- Numbering in Python starts at 0 (not 1)
- Sub-selection in Python is done by indicating the first element you want “up to” but NOT included the last element, negative indexing is possible. Examples:
  - A = [1,2,3,4,5]
  - B=A[0] ; print B # returns 1
  - B=A[2:4]; print B # [3,4]
  - B=A[:3] ; print B # [1,2,3]
  - B=A[3:] ; print B # [4,5]
  - B=A[-1] ; print B # [5]
  - B=A[-3:] ; print B # [3,4,5]
  - B=A[:-3] ; print B # [1,2]
  - B = A[2,-1] ; print B # [3,4]
- List can be altered and extended or shrunk
  - A = [1,2,3,4,5]
  - A.append(6) ; print A # [1,2,3,4,5,6] # Note I operated on A itself, it did NOT return anything
  - A.insert(4,4.5) ; print A # [1,2,3,4,4.5,5,6]
  - A.pop(4) ; print A # [1,2,3,4,5,6]
  - A[-1] = 6.5 ; print A # [1,2,3,4,5,6.5]
- List/tuples elements do NOT have to be of same type
  - A[-1] = (1,2,3) ; print A # [1,2,3,4,5,(1,2,3)]
- Tuples cannot be altered
  - A=(1,2,3) ; A[0]=0

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Python Basics: Important types

- Dictionaries {} are a very useful type. They are a collections of “keys” and “items” associated with each key. Dictionaries are NOT ordered
  - D = { ‘alpha’ : ‘first’, ‘omega’ : ‘last’ }
  - D[‘omega’] # returns ‘last’
- Keys and items can be of ANY type

```
>>> d = {1 : 'one' , 'uno' : 1 , (1,2,3) : 'a tuple' }
>>> d[1] # returns 'one'
>>> d['uno’] # returns 1
>>> d[(1,2,3)] # returns 'a tuple’
```

- You can “query” dictionaries:
  - k = d.keys() ; print k # [1,(1,2,3),’uno’]
  - v = d.values() ; print v # [‘one’, ‘a tuple’, 1]
  - Note the order is not the same than the one a t creation time.
  - d.has_key(“one”)# returns False
  - d.has_key(“uno”) # returns True
- You can “ensure” that something is returned:
- b= d[‘one’] # raises an exception
- b= d.get(‘one’,’wrong key’) ; print b # returns ‘wrong key’

Python Basics
Strings

```
>>> a = 'string are so easy ! Really! I swear!'
>>> a.lower()
'string are so easy ! really! i swear!'
>>> a.upper()
'STRING ARE SO EASY ! REALLY! I SWEAR!'
>>> a.replace('!','.')
'string are so easy . Really. I swear.'
>>> a.split()
['string', 'are', 'so', 'easy', '!', 'Really!', 'I', 'swear!']
>>> a.split('!')
['string are so easy ', ' Really', ' I swear', '']
>>> '.'.join(a.split('!'))
'string are so easy . Really. I swear.'
>>> a = '   too many spaces before and after    '
>>> a.strip()
'too many spaces before and after'
>>> a.lstrip()
'too many spaces before and after    '
>>> a.rstrip()
'   too many spaces before and after'
>>> '1'.zfill(3)
'001'
```

```
>>> a = 'string are so easy ! Really! I swear!'
>>> a.find("easy")
14
>>> a.find("easier")
-1
>>> a[14:]
'easy ! Really! I swear!'
>>> a.swapcase()
'STRING ARE SO EASY ! rEALLY! i SWEAR!'
>>> a.capitalize()
'String are so easy ! really! i swear!'
```

Python Basics: String Formatting

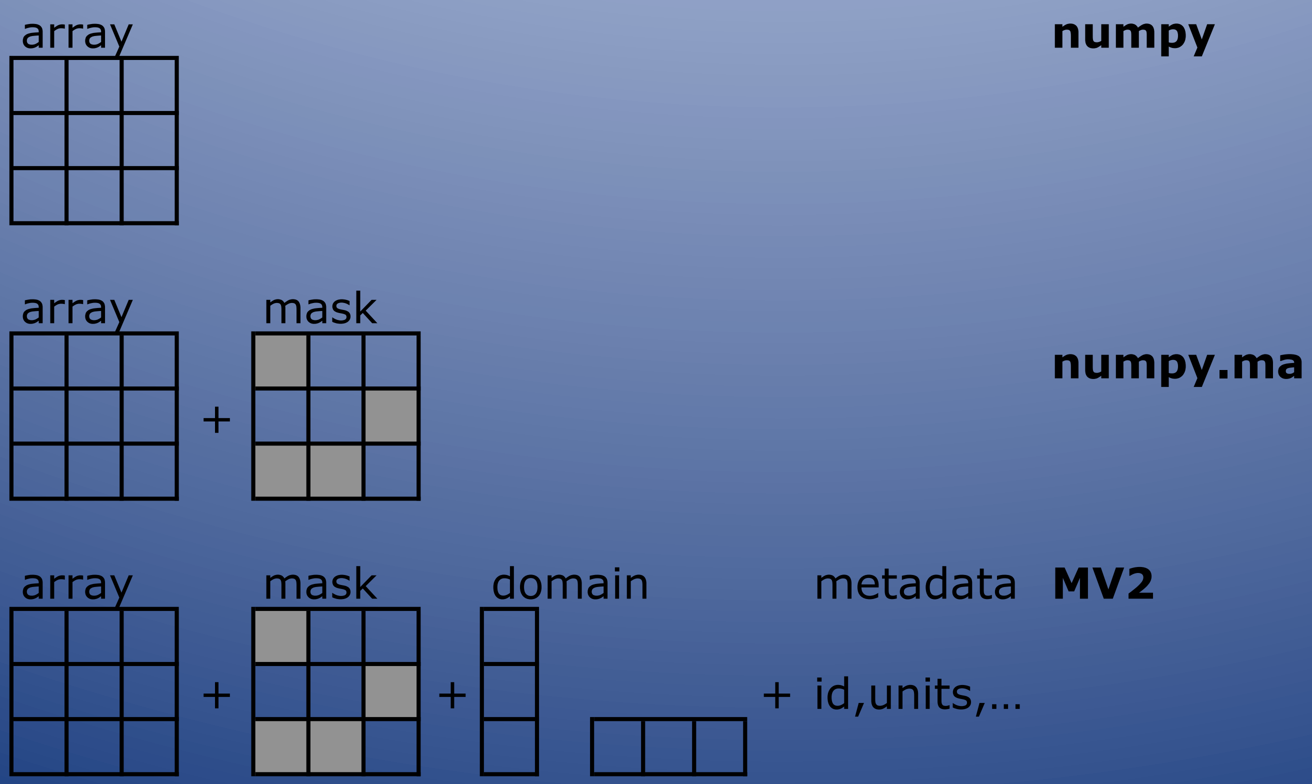
- a=‘my string’
- >>> print ‘%i is a digit\n“%s” is a string\nand %.3f is a float rounded at 3 digits’ % (4,a, 3.14159)

  4 is a digit

  "mystring" is a string

  and 3.142 is a float rounded at 3 digits

Arrays, Masked Arrays and Masked Variables



Arrays, Masked Arrays and Masked Variables

```
>>> a=numpy.array([[1.,2.],[3,4],[5,6]])
>>> a.shape
(3, 2)
>>> a[0]
array([ 1.,  2.])
```

Additional info such as metadata and axes

```
>>>b = MV2.masked_greater(a,4)
>>> b.info()
*** Description of Slab variable_3 ***
id: variable_3
shape: (3, 2)
filename:
missing_value: 1e+20
comments:
grid_name: N/A
grid_type: N/A
time_statistic:
long_name:
units:
No grid present.
** Dimension 1 **
  id: axis_0
  Length: 3
  First:  0.0
  Last:   2.0
  Python id:  0x2729450
** Dimension 2 **
  id: axis_1
  Length: 2
  First:  0.0
  Last:   1.0
  Python id:  0x27292f0
*** End of description for variable_3 ***
```

```
>>> numpy.ma.masked_greater(a,4)
masked_array(data =
 [[1.0 2.0]
 [3.0 4.0]
 [-- --]],
          mask =
 [[False False]
 [False False]
 [ True  True]],
      fill_value = 1e+20)
```

These values are now MASKED (average would ignore them)

I/O
ASCII

- READING

```
F=open("myfile.txt")
Lines = F.readlines()
F.close()
For l in Lines:
  print l
```

```
genutil.ASCII.readAscii.read( text_file ,header=
0, ids=None, shape=None,
next='------',separators=[';',',',':'])
Data in columns:
genutil.ASCII.read_col( text_file ,header=0,
cskip=0, cskip_type='columns', axis=0,
ids=None, idrow=0, separators=[';',',', ':'])
```

- WRITING

```
F=open('myfile.txt',mode) # mode can be "w" or "r+" ("r" is for readonly)
print >> F, 'Hello World'
F.close()
#or
F=open('myfile.txt',mode) # mode can be "w" or "r+" ("r" is for readonly)
Lines =['hello\n','world\n'] # don't forget "\n" at the end of lines
F.writelines(ln)
F.close()
#or
F=open('myfile.txt',mode) # mode can be "w" or "r+" ("r" is for readonly)
Out ='hello world\n'How are you?\n'
F.write(Out)
F.close()
```

I/O
cdms2

- Best way to ingest/write data!
- Opening a file for reading
  - F=cdms2.open(*file_name*)
  - It will open an existing file protected against writing
- Opening a new file for writing
  - F=cdms2.open(*file_name,'w'*)
  - It will create a new file even if it already exists
- Opening an existing file for writing
  - F=cdms2.open(*file_name,'r+'*) # or 'a'
  - It will open an existing file ready for writing or reading

I/O
cdms2

- Multiple way to retrieve data
  - All of it, omitted dimensions are retrieved entirely
    - s=f('var')
  - Specifying dimension type and values
    - S=f('var', time=(time1,time2))
    - Known types: time, level, latitude, longitude (t,z,y,x)
  - Dimension names and values
    - S=f('var',dimname1=(val1,val2))
  - Sometimes indices are more useful than actual values
    - S=f('var',time=slice(indice1,indice2,step))

I/O
cdms2

- Special Case: Time dimension
  - Raw values are not necessarily meaningful
    - 1841664.00 hours since 1800 is actually Feb 5th 2010
  - 2 Solutions
    - Use strings as “value”
      - S=f(var,time=(‘2010’,’2010-2-5 10:30:0.0’))
    - Use cdtime object (see cdms2 doc)
      - T1=cdtime.comptime(2010)
      - T2=cdtime.comptime(2010,2,5,10,30)
      - S=f(var,time=(T1,T2))

cdms2: digression: cdtime

- C = cdtime.comptime(2010,2,5)
- R = C.torel("days since 2010")
- C and R can be passed to MVs for time selection.
- MVs with time axis can have their axis converted to component or relative time.
- T=slab.getTime(); Tc=T.asComponentTime()
- T.toRelativeTime("months since 1800") #actually converts the axis.

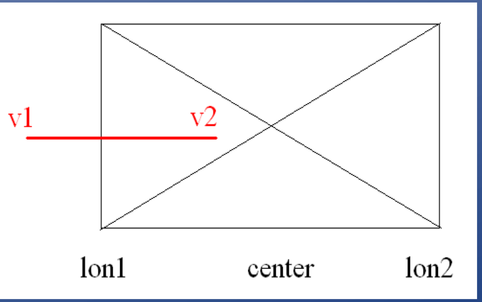
```
>>> t=s.getTime()
>>> t[:2]
array([ 0.,  1.])
>>> tc=t.asComponentTime()
>>> tc[:2]
[1979-1-1 0:0:0.0, 1979-2-1 0:0:0.0]
>>> tr=t.asRelativeTime()
>>> tr[:2]
[0.00 months since 1979-1-1 0, 1.00 months since 1979-1-1 0]
>>> t.toRelativeTime('days since 1979')
>>> t[:2]
array([  0.,  31.])
```

I/O
cdms2 and the mysterious "third argument"

- OK, we understood s=f('var',time=(t1,t2))
- But what's the heck is this mysterious 3rd argument defaulted to '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] #US notation: [v1,v2]
    - » 'co' : [v1,v2[#US notation: [v1,v2)
    - » 'oo' :]v1, v2[#US notation: (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 are within the interval defined
    - In the example on the right:

      (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

I/O

- Other known keywords for data ingestion:
  - squeeze=0/1 # deletes dimensions of length 1
  - order='…zyxt(mydim)…' # Reorders the data
  - cdms selectors
    - cdutil.region predefined (such as cdutil.region.NH)
    - genutil.picker
    - cdms2
      - *from cdms2.selectors import Selector*
      - *sel = Selector(time=('1979-1-1','1979-2-1'), level=1000.)*
      - *x1 = v1(sel)*
      - *x2 = v2(sel)*
  - required
  - raw
  - grid

I/O

- Writing data with cdms2

```
F=cdms2.open("myout.nc","w")
F.write(s)
F.close()
```

- By default dumps NetCDF4 "CLASSIC" compressed. To get Netcdf3:

```
cdms2.setNetcdfDeflateFlag(0) #0/1 (off/on)
cdms2.setNetcdfDeflateLevelFlag(0) #compression level 0 (none) to 9 (max)
cdms2.setNetcdfShuffleFlag(0) #0/1 (off/on)
```

- If dim 0 is time, then variable is extendable

I/O

- Climate Model Output Rewriter: cmor
- AR5 tool to provide model data.

```
import cmor
cmor.setup(inpath='../trunk/Tables')
cmor.dataset('historical', 'ukmo', 'HadCM3', '360_day',model_id='pcmdi-10b')
cmor.load_table('CMIP5_Amon')
id1 = cmor.axis(table_entry='time',units='days since 2000-01-01 00:00:00',
        coord_vals=[15], cell_bounds=[0, 30])
id2 = cmor.axis(table_entry='latitude', units='degrees_north',
        coord_vals=[0], cell_bounds= [-1, 1])
id3 = cmor.axis(table_entry='longitude', units='degrees_east',
        coord_vals=[90], cell_bounds': [89, 91])
axis_ids = [id1,id2,id3]
varid = cmor.variable('ts', 'K', axis_ids)
cmor.write(varid, [273])
path=cmor.close(varid, file_name=True)
```

Data Analysis

- regriding
  - Lat/lon : s.regrid(s2.getGrid())
  - Irregular grids: can take advantage of scrip regridder but need to provide weights
  - coming up (2010 Q3): gridspec regridder

Data Analysis

- numpy (and numpy.ma and MV2) provides an incredibly rich set of ressources for array manipulation, including, but limited to: discrete fourier transform, linear algebra, random sampling, sorting and searching, logical functions, window function, etc…
  - See: http://docs.scipy.org/doc/numpy/reference
- scipy is a set of, mostly, FORTRAN routines used to scientific computation, including, but not limited to: fourier transform, interpolation, optimization, signal processing, linear algebra, sparce matrices and linear algebra, image manipulation, i/o
  - See: http://docs.scipy.org/doc/scipy/reference

Data Analysis: genutil

- genutil.statistics: set of basic statistical functions
- genutil.grower: adding extra dimensions to an array (for example time to a land/sea mask)
- genutil.colors: matching colors to strings:
  - Genutil.colors.str2rgb("orange") # returns: [255,165,0]
- genutil.filters: work in progress, so far only smooth121, custom1D and runningaverage. No options for padding at beg and end yet.
- genutil.picker: cdms2 selector to extract non-contiguous axis values (e.g level 1000 and level 10)

Data Analysis: genutil

- UNIDATA/UDUNITS Python Object
  - initialization: a=unidata.udunits(value,units)
  - a=unidata.udunits(5,'m')
  - b=unidata.udunits(6,'in')
  - c=a+b # udunits(5.1524,"m")
- CONVERSION
  - a.units='feet‘ ; print a # 16.4041994751 feet
  - c=a.to('km') # udunits(0.005,"km")
  - c=unidata.udunits(7,'K') ; factor, offset = c.how('degF') # (1.8, -459.67)
- WHICH UNITS ?
  - lst = c.available_units() # returns list of all known units
  - dict = c.known_units() # dictionary: units (keys) / type (values)
  - dict['k'] # returns : 'THERMODYNAMIC TEMPERATURE'
  - dict = c.known_units(bytype=1) # returns a dictionary of units type (keys) associated with a list of units for each type
  - dict['THERMODYNAMIC TEMPERATURE'] # ['degree_Kelvin', 'degree_Celsius', …]

Data Analysis: genutil

- genutil.statusbar
  - For long script with loops or incremental steps it might be usefull to know if how far along you are.

```
for i in range(1000):
  a=genutil.statusbar(i+1.,1000.)
```

  - Sometimes you might want a graphical bar

```
prev=-1
for i in range(1000):
  prev=genutil.statusbar(i+1.,1000.,prev=prev, tk=1)
```

Data Analysis: cdutil (MV aware)

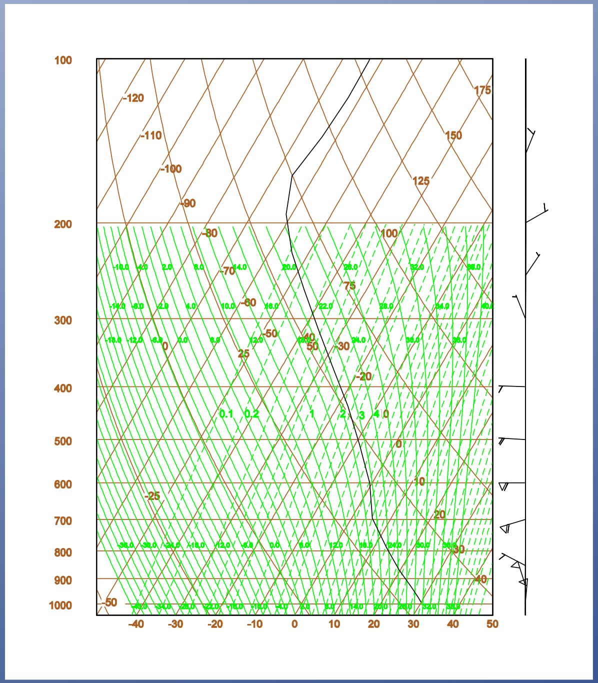
- Set of tools specific to climate data.
- cdutil.averager
  - Area weighted average, can average over multiple dimensions at once, can receive weights as input
- cdutil.region
  - cdms2 selector to extract "exact" region (i.e reset bounds correctly so averaging account for only "actual" area averaged not the full cell.
- cdutil.VariableMatcher
  - Pre-processing of data to compare data on different grids, with different mask and time model.

Data Analysis: cdutil (MV aware)

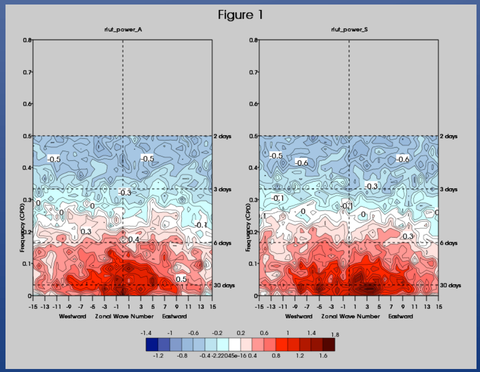
- Set of tools specific to climate data.
- cdutil.times
  - Climatology, Departures, Anomalies Tools works on BOUNDS, NOT on time values, designed for monthly seasons, but one could create an engine for other kind of data (daily, yearly, etc…).
  - In order to set bounds you can use:
    - » cdutil.setTimeBoundsMonthly(Obj)
    - » cdutil.setTimeBoundsYearly(Obj)
    - » cdutil.setTimeBoundsDaily(Obj, frequency=1)
    - Obj can be slab or time axis
  - Create your own seasons:
    - DJFM=cdutil.times.Seasons('DJFM')
- cdutil.vertical
  - Allows for vertical interpolation
  - cdutil.vertical.reconstructPressureFromHybrid
    - Given PS, A, B, P0: P=B*Ps+A*Po
  - cdutil.vertical.linearInterpolation(S,I,levels
    - Given S, I(i.e. Pressure/Depth) : Makes linear interpolation to levels
  - cdutil.vertical.logLinearInterpolation(S,I,levels)
    - Given S, I(i.e. Pressure/Depth) : Makes log-linear interpolation to level

Data Analysis:
PCMDI specialized tools

- thermodynamic diagrams
  - import thermo
  - th=thermo.Gth(x=x,name='test')
  - Entirely customizable
  - Lines/fills are vcs graphic method
  - Can define your own T,P -> X,Y relation
  - Plotting
    - th.plot(t,template=tmpl)
    - T is 1D and axis represents pressure
    - th.plot_windbarb(u,v,P=p)



- Wheeler Kiladis' space-time analysis and plotting.



Data Analysis: other tools worth mentioning

- pyclimate
- natgrid, spherepack, csgrid, regridpack, shgrid, dsgrid.
- ZonalMeans: fortran90 code to compute zonal means on irregular grids.
- eof: Ben’s routines
- MSU: Ben’s original MSU routines.
- Rpy: if you have R then you can call it from python.

Visualization: VCS

- The concepts you need to get to fully understand vcs
- Canvas: x=vcs.init() is the thing on which you draw (your piece of paper)
- Template: WHERE you draw each elements (such as the data area, the legend, the title, etc…)
- Graphic Method: HOW you draw the elements, i.e. boxfill vs isofill, colors and levels to use, labels to use etc…
- Data: WHAT you draw: mainly the array and its mask but also its attribute (name, comments, axes, etc…)

Visualization: VCS

- Available graphic methods:
  - boxfill, isofill, meshfill (irregular grid), isoline,yxvsx (y(x)) , xyvs (x(y)), xvsy, scatter, vector, taylordiagram
  - Example:
    - X=vcs.init()
    - B = X.createboxfill()
    - B.list()
- ALL vcs object have a .list() method that will show which attributes can be set.

VCS: template manipulation

- Template ratio
  - X.ratio=2 : y is twice as big as x
  - X.ratio='auto'
  - X.ratio='2t' : also moves tick marks
- Template scaling (lower left data area unchanged)
  - T.scale(.5) # half size
  - T.scale(.5, axis='x') #half size in X, font unchanged
  - T.scale(.5, axis='x', font=1) # also alter fonts
- Template moving
  - T.move(.2, .4) # move by 20% in x, 40% in y
    - Positive values means up/right
  - T.moveto(x,y) # move lower left corner of data to x,y

VCS: template manipulation

- Using EzTemplate

```
from vcsaddons import EzTemplate
import vcs
x=vcs.init()
M=EzTemplate.Multi(rows=4,columns=3)
for i in range(12):
  t=M.get()
  x.plot(s,t,iso)
```

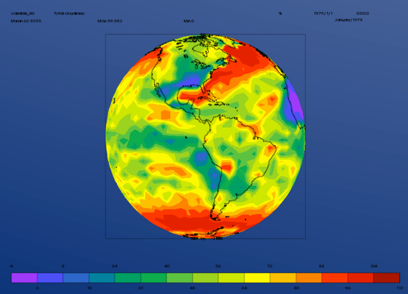
- Also available: EzTemplate.oneD, for 1D plots, uses the provided “base” template and move the legend according to the number of dataset

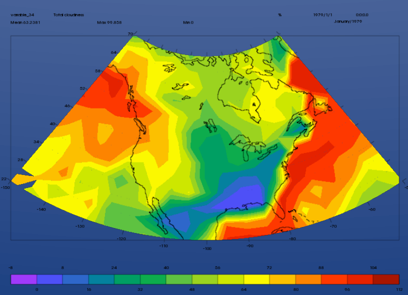
```
OD = EzTemplate.oneD(n=n,template=t)
for i in range(n):
   y = MV2.sin((i+1)*x)
   y.setAxis(0,ax)
   yx = X.createyxvsx()
   yx.linecolor=241+i
   yx.datawc_y1=-1.
   yx.datawc_y2=1.
   t = OD.get()
   X.plot(y,t,yx,bg=bg)
```

VCS: projections

- P=x.createprojection()
- Graphicmethod.projection=P
- P.type=n
  - N can be one of 28 possible
    - print P.__doc__
  - Each type has specific parameters
    - P.list()

```
import vcs,cdms2,sys
f=cdms2.open(sys.prefix+\
  '/sample_data/clt.nc')
s=f("clt",time=slice(0,1),\
   longitude=(-210,50))
x=vcs.init()
iso=x.createisofill()
p=x.createprojection()
p.type='orthographic'
iso.projection=p
x.plot(s,iso,ratio='1t')
x.clear()
p.type='lambert'
x.plot(s(latitude=(20,70),\
longitude=(-150,-50)),iso)
```





vcs: text primitives

- text=x.createtext()

```
----------Text Table (Tt) member (attribute) listings ----------
Tt_name = new
font = 1
spacing = 2
expansion = 100
color = 1
priority = 1
string = None
viewport = [0, 1, 0, 1]
worldcoordinate = [0, 1, 0, 1]
x = None
y = None
projection = default
----------Text Orientation (To) member (attribute) listings ----------
To_name = new
height = 14
angle = 0
path = right
halign = left
valign = half
```

- Font_name = x.addfont(path_to_ttf_font)
- Font = x.getfont(Font_name) # usable in template
- Available fonts by default: ['Adelon', 'Arabic', 'AvantGarde', 'Chinese', 'Clarendon', 'Courier', 'Greek', 'Hebrew', 'Helvetica', 'Maths1', 'Maths2', 'Maths3', 'Maths4', 'Russian', 'Times', 'default']

vcs: other primitives

- fa=x.createfillarea('new')
- l=x.createline('new')
- m=x.createmarker('new')
- Each primitive has the 2 following attributes:
  - Prim.viewport=[xv1,xv2,yv1,yv2] # default: [0,1,0,1]
    - In % of page, area of the primitive extends
  - Prim.worldcoordinates = [x1,x2,y1,y2] # defalut [0,1,0,1]
    - Coordinates corresponding to xv1,xv2,yv1,yv2
    - Primitive units are in the worldcoordinate system
  - Example
    - text.viewport=[.25,.75,.25,.75] # define smaller zone on page
    - text.worldcoordinate=[-180, -90, 180, 90] # Define the coordinate system
    - text.x=[-122.4428]
    - text.y=[37.7709]
    - text.string=['San Francisco, CA, 94117']
- For overlay with an existing graphic method
  - Prim.viewport # set to your template.data
  - Prim.worldcoordinates # set to your graphic method.data.wc

vcsaddons

- Sets of python buit extensions to vcs and also containers so you can easily “extend” vcs.
- Exsisting:
  - Histograms: h=vcsaddons.createhistogram(x=x);x.plot(data,h)
  - Yxvsxfill (filling between 2 curves): y=vcsaddons.createyxvsxfill(x=x);x.plot(d1,d2,y)

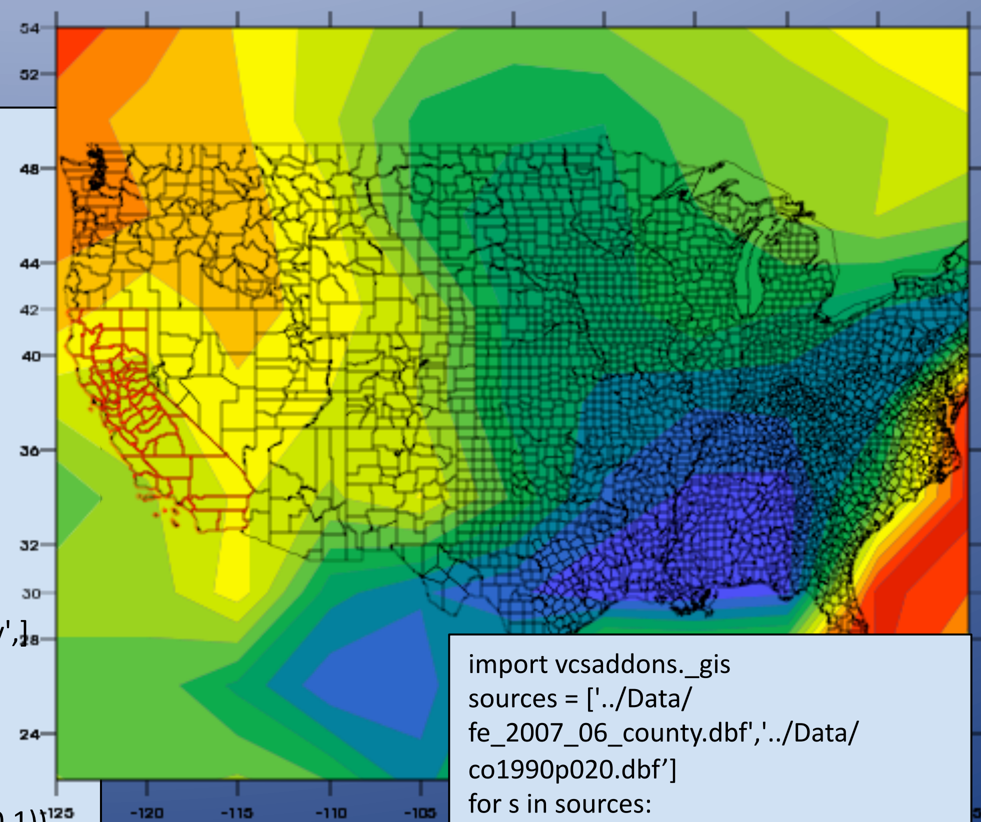
vcsaddons

- GIS capability. You can read and plot gis/shapefiles.

```
import vcs,vcsaddons
import cdms2,sys
x=vcs.init()
import vcs.test.support
bg=0
c=vcsaddons.createusercontinents(x=x)

lon1=-125
lon2=-75.
lat1=20.
lat2=55.

c.types = ['shapefile','shapefile']
c.sources = ['../Data/co1990p020','../Data/fe_2007_06_county',]
c.colors = [246,241,244,241]
c.widths=[1,1,1]
c.lines=['solid','solid','solid','dot']
f=cdms2.open(sys.prefix+’/sample_data/clt.nc’)
s=f("clt",latitude=(lat1,lat2),longitude=(lon1,lon2),time=slice(0,1))
t=x.createtemplate()
iso=x.createisofill()
x.plot(s,t,iso,continents=0,ratio='autot',bg=bg)
x.plot(s,c,t,ratio='autot',bg=bg)
x.png('uscounties')
```




```
import vcsaddons._gis
sources = ['../Data/fe_2007_06_county.dbf','../Data/co1990p020.dbf’]
for s in sources:
    D = vcsaddons._gis.readdbffile(s)
    print D.keys()
    try:
        print D['NAME']
    except:
        print D['COUNTY']
```

Mixing with other languages: Samples source codes

myc.c

```c
float cadd2(float a, float b) {
  return a+b;
}
void cadd(float *a, float *b, float *c) {
  *c = (*a+*b);
  return;
}
void cadd_array(float *a, float *b, float *c, int n) {
  int i;
  for (i=0;i<n;i++) {
    c[i]=a[i]+b[i];
  }
  return;
}
```

myfortran.f90

```fortran
subroutine fadd(a,b,c)
  real a,b,c
  c = a + b
end subroutine fadd
function fadd2(a,b) result (c)
  real a,b,c
  c = a + b
end function fadd2
subroutine fadd_array(a,b,c,n)
  real a(n),b(n),c(n)
  integer n,i
  do i=1,n
    c(i)=a(i)+b(i)
  enddo
end subroutine fadd_array
```

Mixing with other languages: ctypes

-Step 1-
Create a shared library

```
gfortran -c myfortran.f90
gcc -c myc.c
gcc -shared -o mylib.so myfortran.o myc.o
nm mylib.so
```

-Step 1b-
Check it worked

```
nm mylib.so

mylib.so(single module):
00000e98 t __dyld_func_lookup
00000000 t __mh_dylib_header
00000f94 T _cadd
00000f7a T _cadd2
00000fb5 T _cadd_array
00000ec4 T _fadd2_
00000ea6 T _fadd_
00000ef6 T _fadd_array_
00001000 d dyld__mach_header
00000e84 t dyld_stub_binding_helper
```

fortran calls have a "_" Added. This might change depending on machine and compiler

Mixing with other languages: ctypes

-Step 2-
Call from Python

mypython.py

```
import ctypes
mylib = ctypes.CDLL("mylib.so”)
a = ctypes.c_float(2.5)
b = ctypes.c_float(3.)
mylib.cadd2.restype = ctypes.c_float
c =  mylib.cadd2(a,b)
print c
d = ctypes.c_float()
mylib.cadd(ctypes.byref(a),ctypes.byref(b),ctypes.byref(d))
print d.value
e = ctypes.c_float()
mylib.fadd_(ctypes.byref(a),ctypes.byref(b),ctypes.byref(e))
print e.value
mylib.fadd2_.restype = ctypes.c_float
print mylib.myadd2_(ctypes.byref(a),ctypes.byref(b))
```

This section shows how to pass an array

```
a=numpy.arange(10,dtype=numpy.float32)
b=numpy.arange(10,dtype=numpy.float32)
c=numpy.zeros(10,dtype=numpy.float32)
f=numpy.zeros(10,dtype=numpy.float32)
n = ctypes.c_int(10)
mylib.cadd_array(a.ctypes.data_as(ctypes.c_void_p),
            b.ctypes.data_as(ctypes.c_void_p),
            c.ctypes.data_as(ctypes.c_void_p),
            n)
print c
mylib.fadd_array_(a.ctypes.data_as(ctypes.c_void_p),
            b.ctypes.data_as(ctypes.c_void_p),
            f.ctypes.data_as(ctypes.c_void_p),
            ctypes.byref(n))
print f
```

Mixing with other languages: f2py (FORTRAN)

-Step 1-
Optionally alter FORTRAN code to reflect I/O

myfortran.f90

```
subroutine fadd(a,b,c)
  real a,b
  real, intent(out) :: c
  c = a + b
end subroutine fadd
```

-Step 2-
Let f2py do its magic

```
f2py –m mylib –c myfortranlib.f90
```

-Step 3-
Run python

mypython.py

```
import numpy,mylib
a = 2.5
b = 3.
e=0.
e = mylib.fadd(a,b)
print e
print mylib.fadd2(a,b)
a=numpy.arange(10,dtype=numpy.float32)
b=numpy.arange(10,dtype=numpy.float32)
f=numpy.zeros(10,dtype=numpy.float32)
n = 10
mylib.fadd_array(a,b,f,n)
print f
```

Mixing with other languages: Wrapping into Python

- It is nice to be able to call C/FORTRAN but it is nicer to be able to take advantage of Python's strength.
- Let's consider the "add_array" case. Let's assume we have 2 MVs that we would like to add together, but they both have missing data.
- Wouldn't it be nice to create a simple python layer that will retain the C speed and functionalities while pre-processing everything automatically for us and preserving the metadata?

Wrapping with python

- Let's consider the simple function that would do such pre-processing for us

```
import numpy,mylib,MV2,cdms2
def pyadd(a,b):
   """ sums a and b """
   # Add some simple checks
   A = a.astype(numpy.float32)
   B = b.astype(numpy.float32)
   if A.shape!=B.shape:
      raise Exception,"Arrays shapes must match"
   # preserve axes for later
   if isinstance(a,cdms2.tvariable.TransientVariable):
      axes = a.getAxisList()
      atts = a.attributes
   else:
      atts = None
      axes = None
   #flattens the array since our code takes 1D
   # MV2 to make sure it works even on numpy
   A=MV2.ravel(A)
   B=MV2.ravel(B)
```

```
   m1 = A.mask
   m2 = B.mask
   out =
numpy.ravel(numpy.zeros(a.shape,numpy.float32))
   sum = mylib.fadd_array(A.data,B.data,out)
   if m1 is not None:
      out=numpy.ma.masked_where(m1,out)
   if m2 is not None:
      out=numpy.ma.masked_where(m2,out)
   out.shape=a.shape
   if axes is not None:
      out=MV2.array(out)
      out.setAxisList(axes)
      for att in atts:
         setattr(out,att,atts[att])
   out.id='sum'
   return out
```