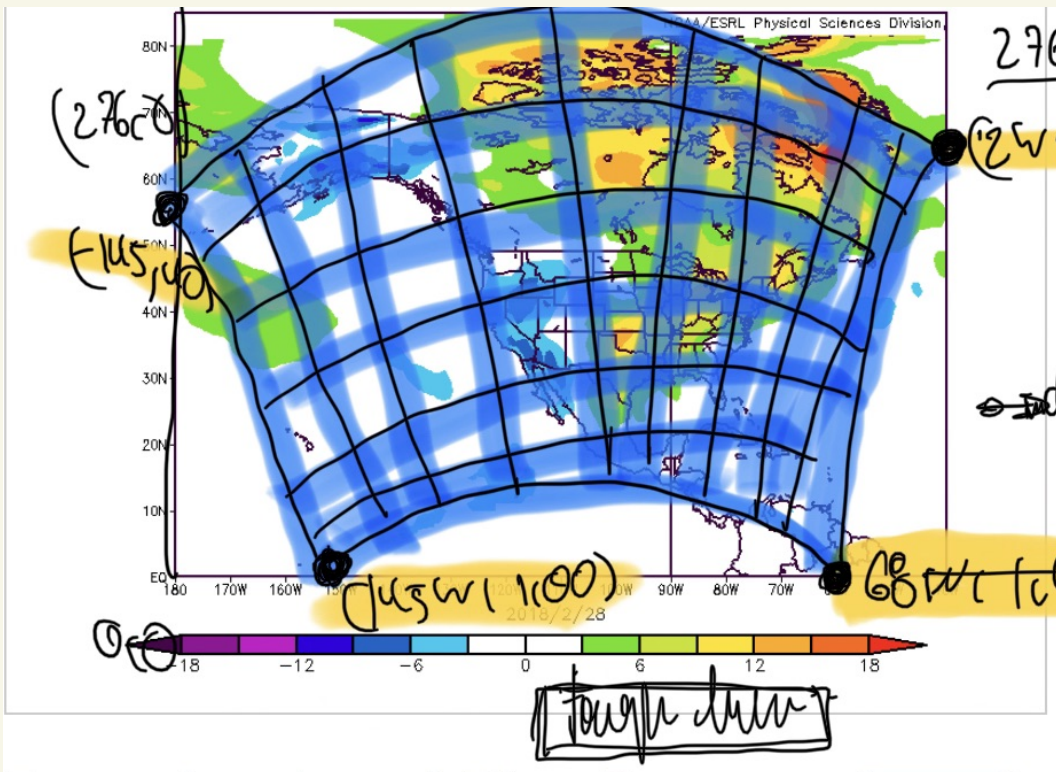
coordinates files in python



Download and Plot Data

Data Help

Search Dataset Variables

Clear

Variable	Statistic	Level	TimeScale	Optio
accumulated snow	Mean	Surface	Daily	

Temporal Coverage:

- 8-times, Daily and Monthly means fo 2022/06/30
- Long Term Daily Monthly means for

Spatial Coverage:

The native model grid is converted t Lambert Conformal Conic grid which archive. Corners of this grid are 1.000001N, 145.5W; 0.897945N, 68. 46.3544N, 2.569891W; 46.63433N, The grid resolution is 349x277 which 0.3 degrees (32km) resolution at the A page describing the coverage alon information on reading the projectio

Levels:

- 29 pressure levels (hPa): 1000, 975, 950, 925, 900, 875, 850, 8 725, 700, 650, 600, 550, 500, 450, 40 250, 225, 200, 175, 150, 125, 100
- soil variables: 0 , 10 , 40 , 100, (800)

Reading NetCDF4 Data in Python (codes included)

Utpal Kumar (https://twitter.com/utpalkmr) 8 minute read UTILITIES October 19, 2020

> "
>
> In Earth Sciences, we often deal with multidimensional data structures such as climate data, GPS data. It 's hard to save such data in text files as it would take a lot of memory as well as it is not fast to read, write and process it. One of the best tools to deal with such data is netCDF4.

Introduction

In Earth Sciences, we often deal with multidimensional data structures such as climate data, GPS data. It 's hard to save such data in text files as it would take a lot of memory as well as it is not fast to read, write and process it. One of the best tools to deal with such data is netCDF4. It stores the data in the HDF5 format (https://en.wikipedia.org/wiki/Hierarchical_Data_Format) (Hierarchical Data Format). The HDF5 is designed to store a large amount of data. NetCDF is the project hosted by Unidata Program at the University Corporation for Atmospheric Research (https://en.wikipedia.org/wiki/University_Corporation_for_Atmospheric_Research) (UCAR).

Similar posts

Three-dimensional perspective map of taiwan using gmt and pygmt (codes included)

High-quality maps using the modern interface to the generic mapping tools (codes included)

Gmt tutorial for beginners (codes included)

Writing netcdf4 data using python (codes included)

Pygmt: high-resolution topographic map in python (codes included)

Here, we learn how to read and write netCDF4 data. We follow the workshop by Unidata (https://www.youtube.com/watch?v=w0cMCKGp9RQ). You can check out the website (http://www.unidata.ucar.edu/software/netcdf/) of Unidata.

Writing NetCDF4 Data using Python

NetCDF file format has been designed for storing multidimensional scientific data such as temperature, rainfall, humidity, etc. In this post, we will see how...

Requirements:–

1. Python3: You can install (https://conda.io/miniconda.html) Python3 via the Anaconda platform. I would recommend Miniconda over Anaconda because it is more light and installs only fundamental requirements for Python.

2. NetCDF4 Package: `conda install -c conda-forge netcdf4`

Reading NetCDF data

Now, we are good to go. Let's see how we can read a netCDF data. The netCDF data has the extension of `.nc` .

- Importing NetCDF and Numpy (a Python library that supports large multi-dimensional arrays or matrices):

```
import netCDF4
import numpy as np
```

Now, let us open a NetCDF Dataset object:

```
f = netCDF4.Dataset('../../data/rtofs_glo_3dz_f006_6hrly_reg3.nc')
```

```
In [2]: f = netCDF4.Dataset('../../data/rtofs_glo_3dz_f006_6hrly_reg3.nc')
        print(f)

        <class 'netCDF4._netCDF4.Dataset'>
        root group (NETCDF4_CLASSIC data model, file format HDF5):
            Conventions: CF-1.0
            title: HYCOM ATLb2.00
            institution: National Centers for Environmental Prediction
            source: HYCOM archive file
            experiment: 90.9
            history: archv2ncdf3z
            dimensions(sizes): MT(1), Y(850), X(712), Depth(10)
            variables(dimensions): float64 MT(MT), float64 Date(MT), float32 Depth(Depth), int32 Y(Y), int32 X(X), float32 La
        titude(Y,X), float32 Longitude(Y,X), float32 u(MT,Depth,Y,X), float32 v(MT,Depth,Y,X), float32 temperature(MT,Depth,
        Y,X), float32 salinity(MT,Depth,Y,X)
            groups:
```

Here, we have read a NetCDF file “rtofs_glo_3dz_f006_6hrly_reg3.nc”. When we print the object “f”, then we can notice that it has a file format of HDF5. It also has other information regarding the title, institution, etc for the data. These are known as metadata.

In the end of the object file print output, we see the dimensions and variable information of the data set. This dataset has 4 dimensions: `MT` (with size 1), `Y` (size: 850), `X` (size: 712), `Depth` (size: 10). Then we have the variables. The variables are based on the defined dimensions. The variables are outputted with their data type such as float64 MT (dimension: MT).

Some variables are based on only one dimension while others are based on more than one. For example, “temperature” variable relies on four dimensions – `MT`, `Depth`, `Y`, `X` in the same order.

We can access the information from this object, `f` just like we read a dictionary in Python.

```
print(f.variables.keys()) # get all variable names
```

```
In [4]: print(f.variables.keys()) # get all variable names
        odict_keys(['MT', 'Date', 'Depth', 'Y', 'X', 'Latitude', 'Longitude', 'u', 'v', 'temperature', 'salinity'])
```

This outputs the names of all the variables in the read netCDF file referenced by “f” object.

We can also individually access each variable:

```
temp = f.variables['temperature'] # temperature variable
print(temp)
```

```
temp = f.variables['temperature']  # temperature variable
print(temp)
```

```
<class 'netCDF4._netCDF4.Variable'>
float32 temperature(MT, Depth, Y, X)
    coordinates: Longitude Latitude Date
    standard_name: sea_water_potential_temperature
    units: degC
    _FillValue: 1.26765e+30
    valid_range: [ -5.07860279  11.14989948]
    long_name:   temp [90.9H]
unlimited dimensions: MT
current shape = (1, 10, 850, 712)
filling on
```

The “temperature” variable is of the type float32 and has 4 dimensions – MT, Depth, Y, X. We can also get the other information (meta-data) like the coordinates, standard name, units of the variable. Coordinate variables are the 1D variables that have the same name as dimensions. It is helpful in locating the values in time and space. The unit of temperature variable data is “degC”. The current shape gives the information about the shape of this variable. Here, it has the shape of (1, 10, 850, 712) for each dimension.

We can also check the dimension size of this variable individually:

```
for d in f.dimensions.items():
  print(d)
```

```
for d in f.dimensions.items():
    print(d)
```

```
('MT', <class 'netCDF4._netCDF4.Dimension'> (unlimited): name = 'MT', size = 1
)
('Y', <class 'netCDF4._netCDF4.Dimension'>: name = 'Y', size = 850
)
('X', <class 'netCDF4._netCDF4.Dimension'>: name = 'X', size = 712
)
('Depth', <class 'netCDF4._netCDF4.Dimension'>: name = 'Depth', size = 10
)
```

The first dimension “MT” has the size of 1, but it is of unlimited type. This means that the size of this dimension can be increased indefinitely. The size of the other dimensions is fixed.

For just finding the dimensions supporting the “temperature” variable:

```
temp.dimensions
```

```
temp.dimensions
```

```
('MT', 'Depth', 'Y', 'X')
```

```
temp.shape
```

```
temp.shape
```

```
(1, 10, 850, 712)
```

Similarly, we can also inspect the variables associated with each dimension:

```
mt = f.variables['MT']
depth = f.variables['Depth']
x,y = f.variables['X'], f.variables['Y']
print(mt)
print(x)
print(y)
```

```
mt = f.variables['MT']
depth = f.variables['Depth']
x,y = f.variables['X'], f.variables['Y']
print(mt)
print(x)
print(y)
```

```
<class 'netCDF4._netCDF4.Variable'>
float64 MT(MT)
    long_name: time
    units: days since 1900-12-31 00:00:00
    calendar: standard
    axis: T
unlimited dimensions: MT
current shape = (1,)
filling on, default _FillValue of 9.969209968386869e+36 used

<class 'netCDF4._netCDF4.Variable'>
int32 X(X)
    point_spacing: even
    axis: X
unlimited dimensions:
current shape = (712,)
filling on, default _FillValue of -2147483647 used

<class 'netCDF4._netCDF4.Variable'>
int32 Y(Y)
    point_spacing: even
    axis: Y
unlimited dimensions:
current shape = (850,)
filling on, default _FillValue of -2147483647 used
```

Here, we obtain the information about each of the four dimensions. The “MT” dimension, which is also a variable has a long name of “time” and units of “days since 1900-12-31 00:00:00”. The four dimensions denote the four axes, namely- MT: T, Depth: Z, X:X, Y: Y.

Now, how do we access the data from the NetCDF variable we have just read. The NetCDF variables behave similarly to NumPy arrays. NetCDF variables can also be sliced and masked.

Let us first read the data of the variable “MT”:

```
time = mt[:]
print(time)
```

```
time = mt[:]  # Reads the netCDF variable MT, array of one element
print(time)
```

```
[ 41023.25]
```

Similarly, for the depth array:

```
dpth = depth[:]
print(depth.shape)
print(depth.dimensions)
print(dpth)
```

```
dpth = depth[:] # examine depth array
print(depth.shape)
print(depth.dimensions)
print(dpth)
```

```
(10,)
('Depth',)
[    0.   100.   200.   400.   700.  1000.  2000.  3000.  4000.  5000.]
```

We can also apply conditionals on the slicing of the netCDF variable:

```
xx,yy = x[:],y[:]
print('shape of temp variable: %s' % repr(temp.shape))
tempslice = temp[0, dpth > 400, yy > yy.max()/2, xx > xx.max()/2]
print('shape of temp slice: %s' % repr(tempslice.shape))
```

```
xx,yy = x[:],y[:]
print('shape of temp variable: %s' % repr(temp.shape))
tempslice = temp[0, dpth > 400, yy > yy.max()/2, xx > xx.max()/2]
print('shape of temp slice: %s' % repr(tempslice.shape))
```

```
shape of temp variable: (1, 10, 850, 712)
shape of temp slice: (6, 425, 356)
```

Now, let us address one question based on the given dataset. “What is the sea surface temperature and salinity at 50N and 140W?“

Our dataset has the variables temperature and salinity. The “temperature” variable represents the sea surface temperature (see the long name). Now, we have to access the sea-surface temperature and salinity at a given geographical coordinates. We have the variables latitude and longitude as well.

The X and Y variables do not give the geographical coordinates. But we have the variables latitude and longitude as well.

```
lat, lon = f.variables['Latitude'], f.variables['Longitude']
print(lat)
print(lon)
print(lat[:])
```

```
lat, lon = f.variables['Latitude'], f.variables['Longitude']
print(lat)
print(lon)
print(lat[:])
```

```
<class 'netCDF4._netCDF4.Variable'>
float32 Latitude(Y, X)
    standard_name: latitude
    units: degrees_north
unlimited dimensions:
current shape = (850, 712)
filling on, default _FillValue of 9.969209968386869e+36 used

<class 'netCDF4._netCDF4.Variable'>
float32 Longitude(Y, X)
    standard_name: longitude
    units: degrees_east
unlimited dimensions:
current shape = (850, 712)
filling on, default _FillValue of 9.969209968386869e+36 used

[[ 45.77320099  45.77320099  45.77320099 ...,  45.77320099  45.77320099
   45.77320099]
 [ 45.82899857  45.82899857  45.82899857 ...,  45.82899857  45.82899857
   45.82899857]
 [ 45.88470078  45.88470078  45.88470078 ...,  45.88470078  45.88470078
   45.88470078]
 ...,
 [ 78.36325073  78.34516144  78.32701111 ...,  56.40559387  56.36401367
   56.32240677]
 [ 78.38998413  78.37184906  78.35365295 ...,  56.41108322  56.36948395
   56.32785034]
 [ 78.41667938  78.39850616  78.38025665 ...,  56.4165535   56.37493134
   56.33327103]]
```

Great! So we can access the latitude and longitude data. Now, we need to find the array index, say iy and ix such that Latitude[iy, ix] is close to 50 and Longitude[iy, ix] is close to -140. We can find out the index by defining a function:

```
# extract lat/lon values (in degrees) to numpy arrays
latvals = lat[:]; lonvals = lon[:]

# a function to find the index of the point closest pt
# (in squared distance) to give lat/lon value.
def getclosest_ij(lats,lons,latpt,lonpt):
  # find squared distance of every point on grid
  dist_sq = (lats-latpt)**2 + (lons-lonpt)**2
  # 1D index of minimum dist_sq element
  minindex_flattened = dist_sq.argmin()
  # Get 2D index for latvals and lonvals arrays from 1D index
  return np.unravel_index(minindex_flattened, lats.shape)

iy_min, ix_min = getclosest_ij(latvals, lonvals, 50., -140)
print(iy_min)
print(ix_min)
```

```
# extract lat/lon values (in degrees) to numpy arrays
latvals = lat[:]; lonvals = lon[:]
# a function to find the index of the point closest pt
# (in squared distance) to give lat/lon value.
def getclosest_ij(lats,lons,latpt,lonpt):
    # find squared distance of every point on grid
    dist_sq = (lats-latpt)**2 + (lons-lonpt)**2
    # 1D index of minimum dist_sq element
    minindex_flattened = dist_sq.argmin()
    # Get 2D index for latvals and lonvals arrays from 1D index
    return np.unravel_index(minindex_flattened, lats.shape)
iy_min, ix_min = getclosest_ij(latvals, lonvals, 50., -140)
print(iy_min)
print(ix_min)
```

```
122
486
```

So, now we have all the information required to answer the question.

```
sal = f.variables['salinity']
# Read values out of the netCDF file for temperature and salinity
print('%7.4f %s' % (temp[0,0,iy_min,ix_min], temp.units))
print('%7.4f %s' % (sal[0,0,iy_min,ix_min], sal.units))
```

```
sal = f.variables['salinity']
# Read values out of the netCDF file for temperature and salinity
print('%7.4f %s' % (temp[0,0,iy_min,ix_min], temp.units))
print('%7.4f %s' % (sal[0,0,iy_min,ix_min], sal.units))
```

```
 6.4631 degC
32.6572 psu
```

Accessing the Remote Data via openDAP

We can access the remote data seamlessly using the netcdf4-python API. We can access via the DAP protocol and DAP servers, such as TDS.

For using this functionality, we require the additional package “siphon”:

```
conda install -c unidata siphon
```

Now, let us access one catalog data:

```
from siphon.catalog import get_latest_access_url
URL =
get_latest_access_url('https://thredds.ucar.edu/thredds/catalog/grib/
NCEP/GFS/Global_0p5deg/catalog.xml',
 'OPENDAP')
gfs = netCDF4.Dataset(URL)

# Look at metadata for a specific variable
# gfs.variables.keys() #will show all available variables.
print("=========================")
sfctmp = gfs.variables['Temperature_surface']
# get info about sfctmp
print(sfctmp)
print("====================")
```

```
# Look at metadata for a specific variable
# gfs.variables.keys() #will show all available variables.
print("=========================")
sfctmp = gfs.variables['Temperature_surface']
# get info about sfctmp
print(sfctmp)
print("====================")
```

```
=========================
<class 'netCDF4._netCDF4.Variable'>
float32 Temperature_surface(time, lat, lon)
    long_name: Temperature @ Ground or water surface
    units: K
    abbreviation: TMP
    missing_value: nan
    grid_mapping: LatLon_Projection
    coordinates: reftime time lat lon
    Grib_Variable_Id: VAR_0-0-0_L1
    Grib2_Parameter: [0 0 0]
    Grib2_Parameter_Discipline: Meteorological products
    Grib2_Parameter_Category: Temperature
    Grib2_Parameter_Name: Temperature
    Grib2_Level_Type: Ground or water surface
    Grib2_Generating_Process_Type: Forecast
unlimited dimensions:
current shape = (92, 361, 720)
filling off

====================
```

```
# print coord vars associated with this variable
for dname in sfctmp.dimensions:
  print(gfs.variables[dname])
```

```
# print coord vars associated with this variable
for dname in sfctmp.dimensions:
    print(gfs.variables[dname])
```

```
<class 'netCDF4._netCDF4.Variable'>
float64 time(time)
    units: Hour since 2017-10-02T18:00:00Z
    standard_name: time
    long_name: GRIB forecast or observation time
    calendar: proleptic_gregorian
    _CoordinateAxisType: Time
unlimited dimensions:
current shape = (92,)
filling off

<class 'netCDF4._netCDF4.Variable'>
float32 lat(lat)
    units: degrees_north
    _CoordinateAxisType: Lat
unlimited dimensions:
current shape = (361,)
filling off

<class 'netCDF4._netCDF4.Variable'>
float32 lon(lon)
    units: degrees_east
    _CoordinateAxisType: Lon
unlimited dimensions:
current shape = (720,)
filling off
```

Dealing with the Missing Data

```
soilmvar =
gfs.variables['Volumetric_Soil_Moisture_Content_depth_below_surface_l
ayer']
print(soilmvar)
print("================")
print(soilmvar.missing_value)
```

```
soilmvar = gfs.variables['Volumetric_Soil_Moisture_Content_depth_below_surface_layer']
print(soilmvar)
print("================")
print(soilmvar.missing_value)
```

```
<class 'netCDF4._netCDF4.Variable'>
float32 Volumetric_Soil_Moisture_Content_depth_below_surface_layer(time, depth_below_surface_layer, lat, lon)
    long_name: Volumetric Soil Moisture Content @ Depth below land surface layer
    units: Fraction
    abbreviation: SOILW
    missing_value: nan
    grid_mapping: LatLon_Projection
    coordinates: reftime time depth_below_surface_layer lat lon
    Grib_Variable_Id: VAR_2-0-192_L106_layer
    Grib2_Parameter: [  2   0 192]
    Grib2_Parameter_Discipline: Land surface products
    Grib2_Parameter_Category: Vegetation/Biomass
    Grib2_Parameter_Name: Volumetric Soil Moisture Content
    Grib2_Level_Type: Depth below land surface
    Grib2_Generating_Process_Type: Forecast
unlimited dimensions:
current shape = (92, 4, 361, 720)
filling off

================
nan
```

I have done several posts of plotting high-resolution maps with geospatial data in Python and Generic Mapping Tools. For the list of all the mapping tools I have covered, visit collections (https://www.earthinversion.com/ei-collections/#plotting-tutorial).

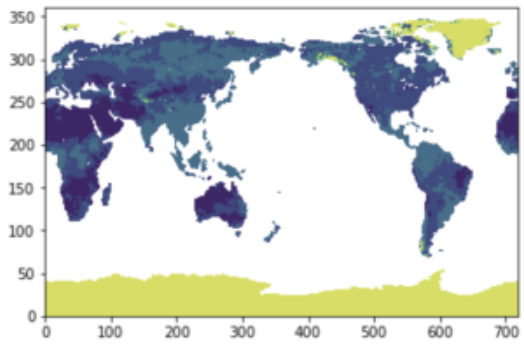
```
# flip the data in latitude so North Hemisphere is up on the plot
soilm = soilmvar[0,0,::-1,:]
print('shape=%s, type=%s, missing_value=%s' % \
 (soilm.shape, type(soilm), soilmvar.missing_value))
```

```
# flip the data in latitude so North Hemisphere is up on the plot
soilm = soilmvar[0,0,::-1,:]
print('shape=%s, type=%s, missing_value=%s' % \
      (soilm.shape, type(soilm), soilmvar.missing_value))
```

```
shape=(361, 720), type=<class 'numpy.ma.core.MaskedArray'>, missing_value=nan
```

```
import matplotlib.pyplot as plt
%matplotlib inline
cs = plt.contourf(soilm)
```

```
import matplotlib.pyplot as plt
%matplotlib inline
cs = plt.contourf(soilm)
```



Here, the soil moisture has been illustrated on the land only. The white areas on the plot are the masked values.

Dealing with Dates and Times

The time variables are usually measured relative to a fixed date using a certain calendar. The specified units are like “hours since YY:MM:DD hh:mm:ss”.

```
from netCDF4 import num2date, date2num, date2index
timedim = sfctmp.dimensions[0] # time dim name
print('name of time dimension = %s' % timedim)
```

```
from netCDF4 import num2date, date2num, date2index
timedim = sfctmp.dimensions[0] # time dim name
print('name of time dimension = %s' % timedim)
```

```
name of time dimension = time
```

Time is usually the first dimension.

```
times = gfs.variables[timedim] # time coord var
print('units = %s, values = %s' % (times.units, times[:]))
```

```
times = gfs.variables[timedim] # time coord var
print('units = %s, \nvalues = %s' % (times.units, times[:]))
```

```
units = Hour since 2017-10-02T18:00:00Z,
values = [   0.    3.    6.    9.   12.   15.   18.   21.   24.   27.   33.   36.
   39.   42.   45.   48.   51.   54.   57.   60.   63.   66.   69.   72.
   75.   78.   81.   84.   87.   90.   93.   96.   99.  102.  105.  108.
  111.  114.  117.  120.  123.  126.  129.  132.  135.  138.  141.  144.
  147.  150.  153.  156.  159.  162.  165.  168.  171.  174.  177.  180.
  183.  186.  189.  192.  195.  198.  201.  204.  207.  210.  213.  216.
  219.  222.  225.  228.  231.  234.  237.  240.  252.  264.  276.  288.
  300.  312.  324.  336.  348.  360.  372.  384.]
```

```
dates = num2date(times[:], times.units)
print([date.strftime('%Y-%m-%d %H:%M:%S') for date in dates[:10]]) #
print only first ten...
```

```
dates = num2date(times[:], times.units)
print([date.strftime('%Y-%m-%d %H:%M:%S') for date in dates[:10]]) # print only first ten...
```

```
['2017-10-02 18:00:00', '2017-10-02 21:00:00', '2017-10-03 00:00:00', '2017-10-03 03:00:00', '2017-10-03 06:00:00',
 '2017-10-03 09:00:00', '2017-10-03 12:00:00', '2017-10-03 15:00:00', '2017-10-03 18:00:00', '2017-10-03 21:00:00']
```

We can also get the index associated with the specified date and forecast the data for that date.

```
import datetime as dt
date = dt.datetime.now() + dt.timedelta(days=3)
print(date)
ntime = date2index(date,times,select='nearest')
print('index = %s, date = %s' % (ntime, dates[ntime]))
```

```
import datetime as dt
date = dt.datetime.now() + dt.timedelta(days=3)
print(date)
ntime = date2index(date,times,select='nearest')
print('index = %s, date = %s' % (ntime, dates[ntime]))
```

```
2017-10-06 13:57:49.056995
index = 30, date = 2017-10-06 15:00:00
```

This gives the time index for a time nearest to 3 days from today, current time.

Now, we can again make use of the previously defined “getcloset_ij” function to find the index of the latitude and longitude.

```
lats, lons = gfs.variables['lat'][:], gfs.variables['lon'][:]
# lats, lons are 1-d. Make them 2-d using numpy.meshgrid.
lons, lats = np.meshgrid(lons,lats)
j, i = getclosest_ij(lats,lons,40,-105)
fcst_temp = sfctmp[ntime,j,i]
print('Boulder forecast valid at %s UTC = %5.1f %s' % \
 (dates[ntime],fcst_temp,sfctmp.units))
```

```
lats, lons = gfs.variables['lat'][:], gfs.variables['lon'][:]
# lats, lons are 1-d. Make them 2-d using numpy.meshgrid.
lons, lats = np.meshgrid(lons,lats)
j, i = getclosest_ij(lats,lons,40,-105)
fcst_temp = sfctmp[ntime,j,i]
print('Boulder forecast valid at %s UTC = %5.1f %s' % \
      (dates[ntime],fcst_temp,sfctmp.units))
```

```
Boulder forecast valid at 2017-10-06 15:00:00 UTC = 304.2 K
```

So, we have the forecast for 2017-10-06 15 hrs. The surface temperature at boulder is 304.2 K.

Simple Multi-file Aggregation

If we have many similar data, then we can aggregate them as one. For example, if we have the many netCDF files representing data for different years, then we can aggregate them as one.

```
!ls ../../data/prmsl*nc
```

```
../../data/prmsl.2000.nc ../../data/prmsl.2004.nc ../../data/prmsl.2008.nc
../../data/prmsl.2001.nc ../../data/prmsl.2005.nc ../../data/prmsl.2009.nc
../../data/prmsl.2002.nc ../../data/prmsl.2006.nc ../../data/prmsl.2010.nc
../../data/prmsl.2003.nc ../../data/prmsl.2007.nc ../../data/prmsl.2011.nc
```

Multi-File Dataset (MFDataset) uses file globbing to patch together all the files into one big Dataset. Limitations:- It can only aggregate the data along the leftmost dimension of each variable.

It can only aggregate the data along the leftmost dimension of each variable. only works with NETCDF3, or NETCDF4_CLASSIC formatted files. kind of slow.

```
mf = netCDF4.MFDataset('../../data/prmsl*nc')
times = mf.variables['time']
dates = num2date(times[:],times.units)
print('starting date = %s' % dates[0])
print('ending date = %s'% dates[-1])
prmsl = mf.variables['prmsl']
print('times shape = %s' % times.shape)
print('prmsl dimensions = %s, prmsl shape = %s' %\
 (prmsl.dimensions, prmsl.shape))
```

```
mf = netCDF4.MFDataset('../../data/prmsl*nc')
times = mf.variables['time']
dates = num2date(times[:],times.units)
print('starting date = %s' % dates[0])
print('ending date = %s'% dates[-1])
prmsl = mf.variables['prmsl']
print('times shape = %s' % times.shape)
print('prmsl dimensions = %s, prmsl shape = %s' %\
     (prmsl.dimensions, prmsl.shape))
```

```
starting date = 2000-01-01 00:00:00
ending date = 2011-12-31 00:00:00
times shape = 4383
prmsl dimensions = ('time', 'lat', 'lon'), prmsl shape = (4383, 91, 180)
```

Finally, we need to close the opened netCDF dataset.

```
f.close()
gfs.close()
```

To download the data, click here (https://github.com/earthinversion/test-data-netCDF). Next, we will see how to write a netCDF data.

Reading and analyzing NetCDF data with the help of Xarray

```
from netCDF4 import Dataset, num2date
import numpy as np
import xarray as xr
import matplotlib.pyplot as plt


## Reading data
## Data at particular time snapshot (particular day)
ncfile = "3B-DAY.MS.MRG.3IMERG.20140101-S000000-E235959.V06.nc4.SUB.nc4"
f1 = Dataset(ncfile)
# print(f1.variables.keys())


## Assigning variables
lats = f1.variables["lat"][:]
lons = f1.variables["lon"][:]
time = f1.variables["time"]

## Data for only one day
dates = num2date(time[:], time.units)
time_of_data = dates[0].strftime("%Y-%m-%d %H:%M:%S")
print(time_of_data)


Calprcp = f1.variables["precipitationCal"][:]
HQprcp = f1.variables["HQprecipitation"]

hqprc_dimensions = f1.variables["HQprecipitation"].dimensions  # ('time', 'lon', 'lat')

SelHQprc = HQprcp[0, :, :]  # remove the time dimension

ds = xr.Dataset(
    {
        "HQprcp": (("lon", "lat"), SelHQprc),
    },
    {
        "lon": lons,
        "lat": lats,
    },
)

df = ds.to_dataframe()

## Visualize the variations with longitude
```

```
plt.figure()
ds.mean(dim="lat").to_dataframe().plot(marker="o")
plt.savefig("variation_with_longitude.png", bbox_inches="tight")

## Visualize the variations with latitude
plt.figure()
ds.mean(dim="lon").to_dataframe().plot(marker="o")
plt.savefig("variation_with_latitude.png", bbox_inches="tight")
```

raw
://gist.github.com/earthinversion/fac9d87c768ddf9c7730a3bb9db36d27/raw/043352425f68118c1048ba6d3b232c3865fa5b33/analyze_data.py)
analyze_data.py (https://gist.github.com/earthinversion/fac9d87c768ddf9c7730a3bb9db36d27#file-analyze_data-py) hosted with ❤️
by GitHub (https://github.com)

```
## All these packages are required for the program
netCDF4
numpy
xarray

## Data
3B-DAY.MS.MRG.3IMERG.20140101-S000000-E235959.V06.nc4.SUB.nc4
```

raw
//gist.github.com/earthinversion/fac9d87c768ddf9c7730a3bb9db36d27/raw/043352425f68118c1048ba6d3b232c3865fa5b33/requirements.txt)
requirements.txt (https://gist.github.com/earthinversion/fac9d87c768ddf9c7730a3bb9db36d27#file-requirements-txt) hosted with ❤️
by GitHub (https://github.com)

Tags: geospatial data visualization | hierarchical data analysis | netcdf | xarray

Categories: utilities

Created on: October 19, 2020

Disclaimer of liability

The information provided by the Earth Inversion is made available for educational purposes only.

Whilst we endeavor to keep the information up-to-date and correct. Earth Inversion makes no representations or warranties of any kind, express or implied about the completeness, accuracy, reliability, suitability or availability with respect to the website or the information, products, services or related graphics content on the website for any purpose.

UNDER NO CIRCUMSTANCE SHALL WE HAVE ANY LIABILITY TO YOU FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF THE SITE OR RELIANCE ON ANY INFORMATION PROVIDED ON THE SITE. ANY RELIANCE YOU PLACED ON SUCH MATERIAL IS THEREFORE STRICTLY AT YOUR OWN RISK.

LEAVE A COMMENT