Lecture 10: NetCDF, OPeNDAP, Xarray, Cartopy, & PyGMT

Guest Lecture by Ty Janoski, Ph.D.

April 2nd, 2025

NetCDF: Network Common Data Form

Sources for this section:

- [CEDA Archive NetCDF Page](#).
- [UCAR NetCDF Users Guide](#)
- [Data Carpentry for Oceanographers Introduction to netCDF](#)

Introduction to NetCDF

NetCDF is an **interface for accessing and manipulated labeled data arrays**. Although it is not *strictly* a file format, it can be thought of as such. NetCDF files will end with ".nc". Most climate, atmospheric, and ocean model data is distributed at netCDF files.

NetCDF has some advantages:

- Files are *self-describing*, i.e., they contain information about the data structure and attributes *in the files themselves* (called metadata)
- Platform independent & many tools to access the data
- Stored in binary format, which is storage-efficient

NetCDF Structure

Data is stored as **labeled arrays** where each array has the same data type.

NetCDF files will typically have **variables**, **dimensions/coordinates**, and **attributes**.
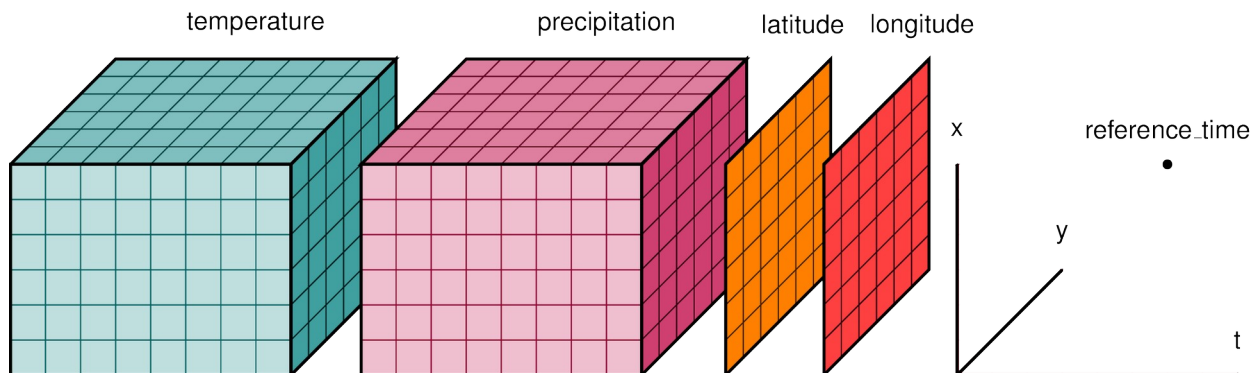
Variables:

Variables are arrays of the same type that store the actual data (i.e., temperature, pressure, salinity). They will have a name, data type, and shape, where the shape corresponds to the dimensions describing the data.
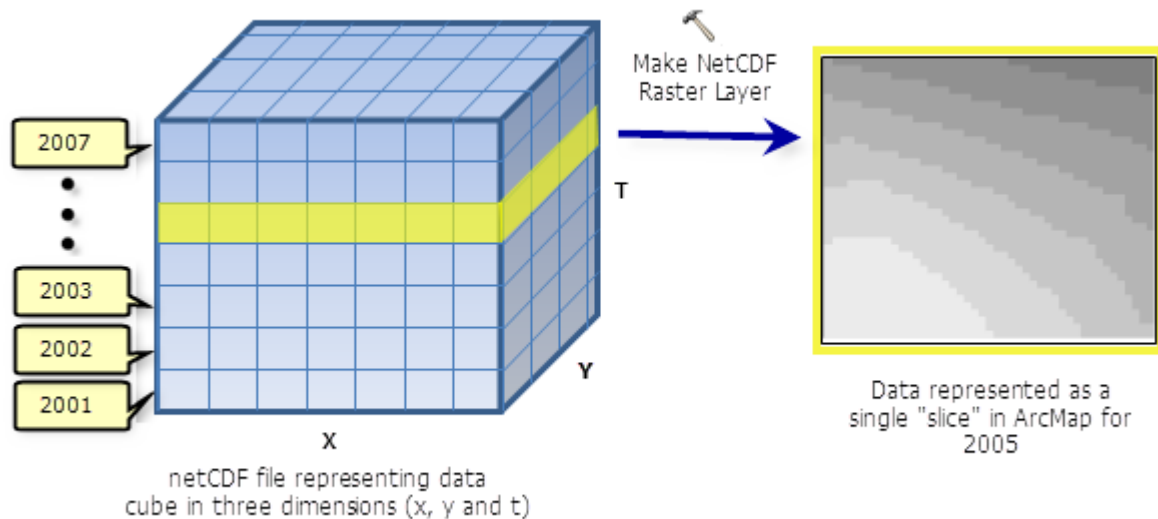
Source: esri ArcGIS blog

Dimensions:

Dimensions provide the shape information for data variables and may represent an actual physical dimension like latitude, longitude, height, or time. They have a name and size.

Coordinate arrays hold data that gives meaning to the dimensions. For example, there might be a "time" dimension of size 365 and a corresponding "time" coordinate array that holds the calendar date for each time.

Attributes:

Attributes *describe* the data such as providing information about map projections, units, or other helpful data properties.

Accessing netCDF data

We will be using OPeNDAP to access output from a climate model and the Python libraries Xarray to interface with it.

OPeNDAP: Open-Source Project for a Network Data Access Protocol

Sources for this section: OPeNDAP Tutorial

OPeNDAP is **a data access protocol that lets you access data remotely**. This means that you do not have to download the data on your machine, which is very helpful for climate data that can be hundreds of TBs. Your computer can likely hold 1 TB.

Note that although we do not have to download the data, depending on what you are doing, you will still have to inevitably *load the data into your memory*. You have to be cognizant of your RAM (typically less than 32 GB on your personal machine) when working with climate data. Using tools like dask can help you get around this, as well as being a smart programmer, but those are beyond the scope of this lecture.

Columbia has a server full of climate model output that is set up so that you can access it with OPeNDAP. All we need is a link and a library, like Xarray, to get started.

Xarray: labelled multi-dimensional arrays

Sources for this section:

- Xarray in 45 minutes
- Ryan Abernathey's Xarray Tutorial
- Project Pythia Introduction to Xarray Cookbook

Introduction & Gathering Data

We will be playing around with model output generated during the *Climate Model Intercomparison Project Phase 6 (CMIP6)*. In this experiment, modeling groups worldwide produced a standard set of simulations with their models.

The first step is to simply import Xarray. We will import NumPy and Matplotlib while we are at it.

```
# Import statements
import xarray as xr
import numpy as np
import matplotlib.pyplot as plt
```

```
# These are some "magic" statements to make displaying plots a bit
smoother
%matplotlib inline
%config InlineBackend.figure_format='retina'
```

Now we are going to "read in" one climate model simulation containing the 2 meter air temperature from one experiment. The model name is CESM (Community Earth System Model) and the scenario is "Historical," meaning that this climate model simulation is forced with historical levels of aerosols, CO2, etc.

Notice how easy it is to use Xarray's `open_dataset()` function to access the data. All we need is the OPeNDAP link!

Slight complication: For whatever reason, the data stored on the Columbia server provided an invalid calendar, so Xarray will get confused when it tries to open it. I am going to provide a `fix_calendar()` function below. Do not worry about the specifics of what is it doing, but realize that in most cases outside of Columbia, you will not have to do this.

```
def fix_calendar(ds: xr.Dataset) -> xr.Dataset:
    """
    Overwrite the dataset's '360' calendar to a '360_day to be CF-
compliant.

    Input:
    ds: the Xarray Dataset whose calendar to fix. Time axis is called
'T'.

    Output:
    fixed: Xarray Dataset with fixed calendar
    """
    ds.T.attrs["calendar"] = "360_day"
    return xr.decode_cf(ds)

# Set URL
url =
("http://mary.ldeo.columbia.edu:81/CMIP6i/.byScenario/.ssp585/.Amon/"+
        ".tas/.CESM2/.gn/.r1i1p1f1/.tas/dods")

# Read in the data while temporarily ignoring the time dimension
ds = xr.open_dataset(url, decode_times=False)
ds = fix_calendar(ds)

# Look at ds
ds
```

Xarray's `open_dataset()` function has several arguments, which is the computer science term for optional parameters. Here are few to know:

- **engine**: specifies the underlying backend to interpret the files. Xarray is "guessing" to use `netcdf4` for the file we opened, but depending on the data, you can alternatively use `h5netcdf` or `zarr` among others
- **decode_cf** and **decode_times**: can set these to false if Xarray is giving you issues trying to understand climate model output.
- **drop_variables**: can give a string or list of strings to specify variables you do not want to read in from the netCDF file.

Datasets & DataArrays

Datasets and DataArrays are the fundamental labeled array containers in Xarray. We read in a Dataset, which contains data variables in the form of individual DataArrays. Look what happens if we pull out the "tas" variable that contains the surface air temperature.

```python
# Extract with dictionary syntax
ds["tas"]

# Extract with dot notation
ds.tas
```

DataArrays have many of the properties we talked about above in the netCDF section.

```python
# Assign tas to a variable so we don't have to keep typing ds.tas
tas = ds.tas

# Print some metadata
print("DataArray name: " + tas.name)
print("DataArray named dimensions: " + str(tas.dims))
print(str(tas.coords))
```

`dims` are our data axes, while `coords` are our labeled coordinate variables. This is an easy example because all of our coordinates are one-dimensional, but this won't always be the case!

Let's also check out the attrs.

```python
tas.attrs
```

We can convert our nicely labeled DataArrays to unlabeled NumPy arrays. This is useful sometimes when we need to perform operations that don't play nice with Xarray or we need to feed the data into another library.

This might take a second because now we are **loading the data into memory**.

```python
tas.data
```

Let's check the data size and type.

```python
type(tas.data)
```

```
# Get size, divide by 10**6 to make it MB
tas.nbytes/1e6
```

That's 228 MB. Pretty big, even for only being a 3-dimensional variable. Can you imagine how big a file containing the vertically varying air temperature with 30+ vertical levels?

Xarray Comparison

Xarray lets us manipulate and visualize data much easier than using the more basic (but important!) libraries like NumPy or Scipy. Consider how we might visualize this data without Xarray.

This is an example of how you should **not** do it!

```
# Get data and coordinates to plot it
lat_data = ds.lat.data
lon_data = ds.lon.data
tas_data = ds.tas.data

# Take the time mean
tas_mean = tas_data.mean(axis=0)

# Now plot it
# The [0,:,:] means we are only plotting the first time
plt.figure()
plt.pcolormesh(lon_data,lat_data,tas_mean)
```

Xarray uses the names and coordinates of variables to perform similar operations and plotting, so it is much more readable.

```
# One line for the same plot
tas.mean(dim="T").plot(x="lon")
```

Look, it even gave us axis labels and a colorbar! Of course we can overwrite any of these properties to beautify the plot however we'd like, but this just shows how much more usable Xarray is. We can clearly see that we took the mean along the "T" (time) axis and plotted it while specifying our x-coordinate is longitude.

Slicing and dicing data

Positional Indexing

Xarray gives us several different ways to subset data, i.e., select which data we want to analyze or plot. The first is similar to how we subset NumPy arrays.

```
# Reminder: tas is our DataArray holding the surface air temperature
with dimensions of T, lat, and lon
# Get the first time
tas[0,:,:]
```

```
# Or a range...
tas[0:5,:,:]
```

Remember: this isn't NumPy, this is **Xarray!** There has to be a better way to index.

Xarray's main advantage is that the arrays are *labeled*. That means instead of memorizing the dimension/coordinate order, we can call them by name using `.isel()`.

```
# Get the first time
tas.isel(T=0)
```

We can also get a range, but it will look a bit different because we will use the `slice()` function.

```
# Get the first five times
tas.isel(T=slice(0,5)) # Note: we can use None instead of 0 here for
the same result
```

This is great, but it requires us to have some prerequisite knowledge of the coordinates. Next, we will show how you can use the `.sel()` function.

Label-based indexing

This is similar to how indexing works in Pandas. We can use `.sel()` to directly reference coordinate names and values!

```
# Get the data just for July of model-year 2016
tas.sel(T="2016-07") # Normally YYYY-MM(-DD) format

# And a range...
tas.sel(T=slice("2016-01","2016-12"))
```

Notice how we we use the Python built-in `slice()` function to specify our slice.

But what if we don't know the *exact* position along a coordinate variable to index our data? Thankfully, Xarray allows us to perform nearest-neighbor sampling.

Say we wanted the data for New York City, which is approximately 40.7☐°N and 74☐°W.

Nearest-Neighbor Sampling

```
# Set the latitude and longitude of nyc
nyc_lat, nyc_lon = 40.7, 360-74 # convert to degrees east

tas_nyc = tas.sel(lat=nyc_lat,lon=nyc_lon,method="nearest")
tas_nyc
```

It's good practice to verify the results of using nearest-neighbor interpolation, but know that there is also a `tolerance` keyword argument if you want to make sure the result is close to what you wanted.

Interpolation

If nearest-neighbor sampling is not enough, we can even interpolate to the exact point of New York City. By default, it will use bilinear interpolate, but there are many other options that can be seen here.

```
# Interpolate to NYC
tas_nyc = tas.interp(lat=nyc_lat,lon=nyc_lon)
tas_nyc
```

Just for fun, let's plot it. It's going to look messy because any signal is **masked by the noise of seasons**. We will address this in the next section.

```
tas_nyc.plot()
```

Computation with Xarray

Arithmetic & NumPy

For the most part, Xarray DataArrays and Datasets work intuitively with arthmetic and NumPy operations.

```
# Convert the temperature from Kelvin to Fahrenheit
tas_F = (tas - 273.15) * (9/5) + 32
tas_F.mean(dim="T").plot()
```

Let's take a step further and calculate the *saturation vapor pressure* from our data! We will use an approximation of the Clausius-Clapeyron relationship. You don't have to worry about how we got this equation, just how we implement it.

$$e_s(T) = 6.112 \times exp\left(\frac{17.67 \cdot T}{T + 243.5}\right) \text{ where T is the temperature in } \square°C$$

```
# Convert to C
tas_C = tas-273.15

# compute saturation vapor pressure
es = 6.112 * np.exp((17.67*tas_C)/(tas_C+243.5))
es.rename("es").mean(dim="T").plot()
```

Notice that even though we called a NumPy function on our DataArray, which in theory should result in a NumPy array as output, it stayed a DataArray!

Broadcasting / Aligning

Broadcasting refers to finding a common array shape when performing operations with arrays that have different shapes. Xarray is *excellent* at broadcasting and does so without much user input, although there are functions like `broadcast()` for explicit broadcasting.

If we multiply the lat and the lon, Xarray automatically creates a new 2-dimensional DataArray with lat and lon dimensions.

```
ds.lat * ds.lon
```

We should also talk about when coordinates are mismatched. Let's create the same DataArray but perturb the latitudes by a very small number.

```
areas = ds.lat * ds.lon
areas["lat"] = areas.lat + 1e-5
```

What happens if we try performing an operation with our actual tas data that has the regular lat?

```
areas * ds.tas
```

Look how we have no latitudes. That's because they didn't share any latitudes in common. In my research experience, this issue has come up a LOT.

Groupby

`groupby()` lets us bin data into groups and reduce. In other words, we can reorganize our data by sorting it into different categories. This is commonly done along the time axis.

```
# groupby the season
tas.groupby("T.season")
```

Notice that using `groupby()` just creates a DataArrayGroupBy object. This data isn't usable in its current form, but notice that it was able to sort it into meteorological seasons. We need to call another function to tell Xarray what to do with the groups.

```
# Get the seasonal averages
tas.groupby("T.season").mean().plot(col="season")
```

Resample

A similar function to `groupby()` is `resample()` which is a special `groupby` for time axes. With this function, we can very easily calculate annual averages.

```
# Get the annual average temperature in NYC
tas_nyc.resample(T="YE").mean().plot()
```

More Types of Plots

Depending on the shape of the data, Xarray can make some pretty nice plots on its own. There are also plenty of keyword arguments.

```python
# Make a line plot of the zonal-average seasonal temperature
tas.groupby("T.season").mean(dim=["lon","T"]).plot.line(hue="season",y
="lat")

# Contour North America's annual average temperatures
tas.sel(lat=slice(20,50),lon=slice(220,300)).mean(dim="T").plot.contou
r(levels=10,cmap="vanimo",add_colorbar=True)
```

Reading and Writing Xarray Datasets/DataArrays

Not all data you will encounter will be in a nice netCDF file pre-labeled for you, but that doesn't stop Xarray from being useful. Let's generate some data and try making a DataArray

```python
# Generate random data
data = 100 * np.random.randn(100,100)

# Try casting it into a DataArray
xr.DataArray(data)
```

There are no dimension names! We can fix this by providing dimension names.

```python
xr.DataArray(data,dims=["lat","lon"])
```

Next step, let's actually try adding coordinate variables. This will be make-believe latitude and longitudes.

```python
# Create latitude and longitude numpy arrays
lat = np.linspace(-90,90,100)
lon = np.linspace(0,360,100)

# Try making a DataArray now
our_da = xr.DataArray(data,dims=["lat","lon"],coords=[lat,lon])
our_da
```

We can add useful attributes, too.

```python
our_da.lat.attrs["units"] = "degree_north"
our_da.lon.attrs["units"] = "degree_east"

our_da.lat
```

What about a Dataset with multiple DataArrays? We can use `xr.merge()` to put DataArrays together.

```python
# make a new DataArray
data2 = 100 * np.random.randn(100,100)
our_other_da = xr.DataArray(data2,dims=["lat","lon"],coords=[lat,lon])
```

```
# Name our DataArrays! Important!
our_da = our_da.rename("dummy_var_1")
our_other_da = our_other_da.rename("dummy_var_2")

# Combine into one Dataset
our_ds = xr.merge([our_da,our_other_da])
our_ds
```

What if we wanted to save or publish this data? We can easily save it as a netCDF file.

```
our_ds.to_netcdf("dummy_netcdf.nc")
```

Xarray Exercise

Let's look at the climatology of the surface air temperature in Brisbane, Australia. Produce a time series with month on the x-axis and surface air temperature in ☐°C on the y-axis.

1. Convert the temperature to ☐°C and *fix the units attribute*.
2. Select the point closest to Brisbane, Australia (27.5°S, 153.0°E).
3. Using `groupby()`, create the monthly climatology of the surface air temperature.
4. Plot it using Xarray's `plot()` function.

```
# Convert to C

# Get data for Brisbane

# Create monthly climatology

# Plot
```

Plotting Geospatial Data with Cartopy

Introduction

We've already seen how we can create nice plots with Xarray alone. However, we need something more robust to put the data *on a map*, making sure that our data is being represented accurately on latitude/longitude grids.

We will learn about two Python libraries for plotting data. The first library is **Cartopy** (Cartography Python), which is older and more widely used than the other library we will discuss, **PyGMT**. Each has their advantages and the choice of which to use is entirely up to you.

Sources for this section:

- [Ryan Abernathey's Cartopy Lecture](#)
- [Project Pythia Introduction to Cartopy Cookbook](#)
- [Using Cartopy with Matplotlib](#)

```python
# We have to start by importing Cartopy :)
# Rarely do we have to import ALL of Cartopy, just some specific
submodules

import cartopy.crs as ccrs # coordinate reference system
import cartopy.feature as cfeature # Cartoy's feature library
(boundaries, etc.)
```

Let's check out some of the projections available to us by typing `ccrs.` and hitting tab.

```python
# ccrs.
```

And look at one projection.

```python
ccrs.PlateCarree?
```

We can call it to see what the map looks like by default with no data.

```python
ccrs.PlateCarree()
```

```python
ccrs.EckertIV()
```

```python
ccrs.NorthPolarStereo()
```

GeoAxes

Cartopy's main draw is how well it interacts with Matplotlib. If you remember from previous lectures, when you create a plot in Matplotlib, you produce an Axes (or AxesSubplot) object. Cartopy overrides this to produce an instance of its own axes class, called GeoAxes. Let's start by making a matplotlib figure with some coastlines. Note that we will call the Matplotlib axes method with the "projection" argument. This tells Matplotlib we are going to be creating a Cartopy GeoAxes object. Let's make a figure with the Mollweide projection.

```
fig = plt.figure() # make a new figure
ax = plt.axes(projection=ccrs.Mollweide()) # create a new GeoAxes
instance
ax.coastlines() # plot the coastlines
```

GeoAxes contains many helpful functions, including many similar to the plotting functions of a Matplotlib Axes instance. We already used `coastlines()`, but let's try a few more.

```
fig = plt.figure() # make a new figure
ax = plt.axes(projection=ccrs.Mollweide()) # create a new GeoAxes
instance

ax.coastlines() # plot the coastlines
ax.stock_img() # Add a land/sea color background
ax.gridlines() # Add gridlines

# Specify the lat/lon box to display in the plot
ax.set_extent([-10,45,35,70]) # Zoom in on Europe
```

Features

We imported `cfeature` before because it is the submodule that lets us access some Natural Earth datasets containing "features" like political boundaries. Let's add a few features to our plot.

```
fig = plt.figure() # make a new figure
ax = plt.axes(projection=ccrs.Mollweide()) # create a new GeoAxes
instance

ax.coastlines() # plot the coastlines
ax.stock_img() # Add a land/sea color background
ax.gridlines() # Add gridlines

# Specify the lat/lon box to display in the plot
ax.set_extent([-10,45,35,70]) # Zoom in on Europe

# Add political borders and rivers
ax.add_feature(cfeature.RIVERS,edgecolor="blue",linewidth=0.5)
ax.add_feature(cfeature.BORDERS)
```

By default, the borders and other features are pretty low-resolution. If we wanted to zoom on an individual country, we would probably want higher resolution features, which we can easily do.

```
# Get higher resolution features
borders_50m =
cfeature.NaturalEarthFeature("cultural","admin_0_countries","50m",face
color="none")
admin1_10m =
```

```
cfeature.NaturalEarthFeature("cultural","admin_1_states_provinces","10
m",facecolor="none",edgecolor="gray",linewidth=0.3)
rivers_10m =
cfeature.NaturalEarthFeature("physical","rivers_lake_centerlines","10m
",edgecolor="deepskyblue",facecolor="none")
ocean_50m =
cfeature.NaturalEarthFeature("physical","ocean","50m",edgecolor="none"
,facecolor=cfeature.COLORS["water"])
land_50m =
cfeature.NaturalEarthFeature("physical","land","50m",edgecolor="none",
facecolor=cfeature.COLORS["land"])

fig = plt.figure() # make a new figure
ax = plt.axes(projection=ccrs.Mollweide()) # create a new GeoAxes
instance

# Specify the lat/lon box to display in the plot
ax.set_extent([6, 19, 36, 48]) # Zoom in on Europe

# Iterate through list of features and add them to our map
# Note: the order is IMPORTANT!
for f in [land_50m,borders_50m,admin1_10m,rivers_10m,ocean_50m]:
    ax.add_feature(f)
```

Combining Data & Cartopy

Projection & Transform

As we start to plot data on our Cartopy GeoAxes, there are two important keyword arguments to keep in mind: **projection** and **transform**.

We've already had practice setting the projection of the GeoAxes object we create. However, if we want to plot data, *we need to give Cartopy more information about what our data looks like*. This is where the transform keyword argument comes in.

Cartopy's documentation has a great explanation. The core concept is that the projection of your axes is independent of the coordinate system your data is defined in. The projection argument is used when creating plots and determines the projection of the resulting plot (i.e. what the plot looks like). The transform argument to plotting functions tells Cartopy what coordinate system your data are defined in.

Let's show this by plotting a line between two points. On a sphere, this can be done a few ways, including:

- as the shortest straight line on the sphere (Geodetic)
- as a straight line in projected (cartesian) space

```
# set the lat and lons of our points
# remember, degrees west and south are negatives
# dictionaries should look familiar :)
```

```
nyc = dict(lat=40.7128,lon=-74.0060)
dubai = dict(lat=25.2048,lon=55.2708)

# make a list of the lats and lons for plotting
lats = [nyc['lat'],dubai['lat']]
lons = [nyc['lon'],dubai['lon']]
```

Now we will plot the points and lines on a PlateCarree projection, once with specifying the transform and once without. Note that if you do not specify the transform, Cartopy will assume the data is formatted the same as the underlying GeoAxes object.

```
ax = plt.axes(projection=ccrs.PlateCarree())
ax.plot(lons, lats, label='Equirectangular straight line')
ax.plot(lons, lats, label='Great Circle', transform=ccrs.Geodetic())
ax.coastlines()
ax.legend()
ax.set_global()
```

As you can see, including the transform argument told Cartopy that our data is defined in the Geodetic coordinate system, and thus we plotted our great circle line.

NumPy arrays with Cartopy

We will make some fake data for now to show how plotting works with basic, unlabeled NumPy arrays. Later, we will combine our model data with Xarray and Cartopy to produce plots.

```
# reminder: the linspace function makes equidistant points
lon = np.linspace(-80, 80, 25) # 80W to 80E
lat = np.linspace(30, 70, 25) # 30N to 70N
lon2d, lat2d = np.meshgrid(lon, lat) # makes 2D grid of lat/lon

# make some random data
data = np.cos(np.deg2rad(lat2d) * 4) + np.sin(np.deg2rad(lon2d) * 4)
```

What happens when we plot it without the transform argument?

```
# The projection keyword determines how the plot will look
plt.figure()
ax = plt.axes(projection=ccrs.PlateCarree())
ax.set_global() # show the entire map
ax.coastlines() # add coastlines
ax.contourf(lon, lat, data) # didn't use transform, but looks ok...
```

This looks fine because we are plotting on a PlateCarree projection and our data is defined in the same coordinate system. Even if we include the transform argument, nothing should change.

```
# The data are defined in lat/lon coordinate system, so PlateCarree()
# is the appropriate choice:
```

```
data_crs = ccrs.PlateCarree()

# The projection keyword determines how the plot will look
plt.figure()
ax = plt.axes(projection=ccrs.PlateCarree())

ax.set_global()
ax.coastlines()
ax.contourf(lon, lat, data, transform=data_crs)
```

Issues arise when we exclude the transform argument and our data and projection are not on the same coordinate system. Look at what happens if we try plotting the data without transform on a different projection.

```
# Now we plot a rotated pole projection
projection = ccrs.RotatedPole(pole_longitude=-177.5,
pole_latitude=37.5)
plt.figure()

ax = plt.axes(projection=projection)
ax.set_global()
ax.coastlines()
ax.contourf(lon, lat, data) # didn't use transform, RIP
```

That's definitely not right. Cartopy is assuming our data is on the rotated pole coordinate system already, which is incorrect. Let's fix it.

```
# A rotated pole projection again...
projection = ccrs.RotatedPole(pole_longitude=-177.5,
pole_latitude=37.5)
plt.figure()

ax = plt.axes(projection=projection)
ax.set_global()
ax.coastlines()

# ...but now using the transform argument
ax.contourf(lon, lat, data, transform=data_crs)
```

Here's our lesson: **Always remember to include the transform and projection arguments when plotting data with Cartopy!**

Xarray & Cartopy

You should already have your surface air temperature from earlier. Let's try plotting it on a map with Cartopy.

```
fig = plt.figure()
ax = plt.axes(projection=ccrs.Mollweide()) # let's go with Mollweide
```

```python
# pick some date to plot the temperature
tas.sel(T="2050-06").plot(ax=ax,
    transform=ccrs.PlateCarree(), # Notice that we can specify the
transformation here
    cbar_kwargs={'shrink': 0.5}, # This is just to shrink the colorbar
a bit
)

# spruce it up
ax.coastlines()
ax.gridlines()

# trim some whitespace
plt.tight_layout()
```

In just a few lines, we were able to create a publication-worthy plot. Cartopy and Xarray work well together.

Example: Plotting Surface Winds

Let's do something a bit different. Let's try looking at the annual average 10 meter winds in CONUS in a CMIP6 experiment.

```python
# Set URL
url_uas =
("http://mary.ldeo.columbia.edu:81/CMIP6i/.byScenario/.ssp585/.Amon/"+
        ".uas/.GFDL-CM4/.gr1/.r1i1p1f1/.uas/dods") # URL to zonal-
wind file
url_vas = url_uas.replace("uas","vas") # Replace uas with vas for
meridional-wind file

# Read in the data while temporarily ignoring the time dimension
ds_uas = fix_calendar(xr.open_dataset(url_uas, decode_times=False))
ds_vas = fix_calendar(xr.open_dataset(url_vas, decode_times=False))

# Combine uas and vas into one Dataset
ds_winds = xr.merge([ds_uas,ds_vas])

fig = plt.figure(figsize=(8,6))

# Specify our projection
proj = ccrs.AlbersEqualArea(central_longitude=-100,
central_latitude=40)
ax = plt.axes(projection=proj)

ds_winds.mean(dim="T").plot.quiver(x="lon",y="lat",u="uas",v="vas",tra
nsform=ccrs.PlateCarree(),ax=ax,scale=100,color="red")

ax.set_extent((-127,-63,20,50))
```

```python
# spruce up the plot, this time with states and borders
ax.add_feature(cfeature.BORDERS)
ax.add_feature(cfeature.STATES)
ax.coastlines()
ax.gridlines()

plt.tight_layout()
```

Exercise: Plotting Arctic Sea Ice

Using the sic variable defined below, create a map of the sea ice concentration in March of 1990*. Use an orthographic projection centered on the North Pole. Add coastlines, gridlines, and be sure to use an appropriate colormap.

* As a reminder, these historical model simulations do not correspond to the actual weather at the given date. You can think of it as March 1990 in some other timeline cooked up by the model.

```python
ds =
fix_calendar(xr.open_dataset("http://strega.ldeo.columbia.edu:81/CMIP5
/.byScenario/.historical/"+

".seaIce/.mon/.sic/.CanESM2/.r1i1p1/.sic/dods",decode_times=False))
ds.sic

# make figure with Orthographic projection

# select and plot sic at correct date

# add coastlines and gridlines
```

Exercise: Plotting ENSO

El Niño and La Niña events are some of the most impactful climate events on Earth. El Niños produce especially warm sea surface temperatures in the Eastern Pacific near Peru and central Pacific, while La Niña events produce anomalously cold sea surface temperatures.

Using the dataset below, calculate the average sea surface temperature anomaly over the period from July 2007 to June 2008. Plot it on a Mollweide projection, centered on the Pacific, and set the extent to be from 95E to 55W and 60S to 70N. Don't forget coastlines. Based on your plot, was the 2007-2008 event an El Niño or La Niña?

```python
sst =
xr.open_dataset("https://psl.noaa.gov/thredds/dodsC/Datasets/noaa.erss
t.v5/sst.mnmean.nc").sst
sst

# calculate anomaly

# make figure and Mollweide projection
```

```
# plot anomaly, set extent, add coastlines
```

Alterative Mapping Library: PyGMT

Sources:

- [PyGMT Documentation](#)

PyGMT is a Python library for making maps and figures using the Generic Mapping Tools, a widely used command-line program in the Earth Sciences. It is a solid alternative to Cartopy for map-making. Some advantages over Cartopy are:

- Higher quality maps
- More functions for plotting geophysical data
- Full integration with Generic Mapping Tools for even more functions
- **Does not use Matplotlib!**

However, there are a few reasons why you might pick Cartopy over PyGMT:

- PyGMT has a steeper learning curve, less Pythonic
- Less flexible for general mapping purposes
- Requires Generic Mapping Tools installed
- Smaller user base

TL;DR: PyGMT is more specialized for the Earth Sciences and tends to make better maps out-of-the-box, but its specialization comes at a cost of usability and generic application.

```
# Start by importing PyGMT
import pygmt
```

Shorelines

Starting easy, let's just plot shorelines.

```
fig = pygmt.Figure() # Make a PyGMT figure
fig.basemap(region="g", projection="W15c", frame=True) # g=global,
W15c=projection
fig.coast(shorelines=True) # Draw shorelines
fig.show()
```

Remember, PyGMT does NOT use Matplotlib. It is its own Figure object.

```
type(fig)
```

There are 4 levels of shorelines, ranging from coastlines to lakeshores, all the way down to lakes in islands in lakes! To only plot the coastlines, we are going to specify the shorelines argument in the `coast()` function with a "1" for only the highest level (coastlines) and the pen to draw them.

```
fig = pygmt.Figure()
fig.basemap(region="g", projection="W15c", frame=True) # btw,
frame=plot outline
fig.coast(shorelines="1/0.5p,black")
fig.show()
```

Let's finish off this section by adding colors for land and water.

```
fig = pygmt.Figure()
fig.basemap(region="g", projection="W15c", frame=True)
fig.coast(land="#666666", water="skyblue") # Specify colors now
fig.show()
```

Frames, Ticks, Titles, and Labels

Previously, we set `frame = True`, but we can actually modify the frame on our plots.

```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60],
projection="M25c") # Mercator
fig.basemap(frame="f") # f = default GMT frame
fig.show(width=800) # Notice that figure width goes here
```

To add ticklines like we did in our previous maps, we set `frame` to either `True` or `"af"`.

```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60],
projection="M25c")
fig.basemap(frame="af")
fig.show(width=800)
```

We can add gridlines. It takes a bit of work to understand the syntax, but we can add gridlines by adding "g" to our `frame` argument. We can also specify the gridlines spacing (in degrees) by adding numbers after `a`, `f`, or `g`.

```
fig = pygmt.Figure()
fig.coast(shorelines="1/0.5p", region=[-180, 180, -60, 60],
projection="M25c")
fig.basemap(frame="a30f7.5g15")
fig.show(width=800)
```

We will end by showing how you can add a title to your plot. Titles get passed to the `frame` argument with a preceding `+t`.

```
fig = pygmt.Figure()
# region="TT" specifies Trinidad and Tobago using the ISO country code
fig.coast(shorelines="1/0.5p", region="TT", projection="M25c")
fig.basemap(frame=["a", "+tTrinidad and Tobago"])
fig.show(width=800)
```

Xarray & PyGMT

In my opinion, Xarray and PyGMT do not work as nicely together as Xarray and Cartopy, but it's getting there. Let's try making a map of our surface air temperature data on a PyGMT map. We will rely on the `grdimage()` function.

```
region = [275, 330, -60, 15]  # Set region to South America

fig = pygmt.Figure() # Create PyGMT Figure object

# Use the grdimage function to plot data on map
fig.grdimage(
    grid=tas.mean(dim="T"),
    region=region,
    cmap="viridis",
    projection="W15c",
    frame=["af","+tAnnual Average"] # Adds a title, too
)

# Add coastlines and borders with intermediate resolution
fig.coast(
    shorelines="1/0.5p,black",
    resolution="i",
    borders="1/0.5p,black")

# Add a colorbar. x+l is our colorbar label, f+l is the units
fig.colorbar(
    frame=['af+l"K"', f"x+lSurface Air Temperature"]
)

# Let's a few cities on our map
# Define cities
cities = [
    {"name": "Bogotá", "lat": 4.711, "lon": 285.928},
    {"name": "São Paulo", "lat": -23.550, "lon": 313.367},
    {"name": "Buenos Aires", "lat": -34.603, "lon": 301.619},
    {"name": "Lima", "lat": -12.046, "lon": 282.958},
    {"name": "Santiago", "lat": -33.448, "lon": 289.331},
]

# Plot each city
for city in cities:
    fig.plot(x=city["lon"], y=city["lat"], style="c0.25c",
```

```
pen="white")
    fig.text(x=city["lon"] + 1, y=city["lat"], text=city["name"],
font="10p,Helvetica-Bold,black", justify="LM")

fig.show(width=500)
```

Exercise: Sea level change with Xarray and PyGMT.

Use the code below to read in a CMIP6 model's sea-level variable. Then, calculate the change in sea level between 2015 and 2100. Finally, plot it using PyGMT. Use the "R15c" projection and the "vik" colormap.

```
import gcsfs

fs = gcsfs.GCSFileSystem(token="anon")

ds =
xr.open_zarr("gs://cmip6/CMIP6/ScenarioMIP/NOAA-GFDL/GFDL-ESM4/ssp585/
r1i1p1f1/Omon/zos/gr/v20180701/", consolidated=True)
ds

# Calculate sea-level change

# Make PyGMT figure and plot it
```

Extra: Get CMIP6 Data from Google Cloud Storage

Source: Pangeo Gallery CMIP6 and Google Cloud Storage Tutorial

So far, we have been accessing CMIP6 data stored on a server located at Lamont-Doherty Earth Observatory. OPeNDAP is great, but science is generally moving towards cloud-based data storage. Rather than purchasing a server with a fixed lifespan, upkeep costs, and unreliable hardware, universities can pay Google or Amazon to host the data for them. It is more reliable and efficient and still does not require us to have the data on our machines. Below, we look at how we can query and access such data.

Let's start by just looking at the data!

```
# Imports
import zarr
import fsspec
import pandas as pd

# Read CSV will read the comma-separated value file on the cloud
df = pd.read_csv('https://storage.googleapis.com/cmip6/cmip6-zarr-
consolidated-stores.csv')
df.head() # prints the headers and a few lines
```

We can use Pandas' `query()` function to find data matching what we'd like. Below, we look for data from the "historical" experiment, monthly atmospheric means (Amon), and downwelling shortwave radiation at the top of the atmosphere.

```python
df_rsdt = df.query("activity_id=='CMIP' & table_id == 'Amon' &
variable_id == 'rsdt' & experiment_id == 'historical'")
df_rsdt
```

We can further refine by finding experiments from NCAR.

```python
df_rsdt_ncar = df_rsdt.query('institution_id == "NCAR"')
df_rsdt_ncar
```

Let's try loading in the data, just the first one on the DataFrame above.

```python
# get the path to a specific zarr store
zstore = df_rsdt_ncar.zstore.values[-1]
print(zstore)

# open it using xarray and zarr
ds = xr.open_zarr(zstore, consolidated=True)
ds
```

And we can use the data exactly like how we were when we were accessing it from OPeNDAP. However, be careful! The dimensions and coordinate names are slightly different.

```python
ds.rsdt.groupby("time.season").mean(dim="time").plot(col="season")
```