

# Xarray

In this lesson we are going to learn about the `Xarray` library - one of the most useful libraries available to us as earth/ocean/atmospheric researchers.

## credit

This lesson is from the Geohackweek tutorial on Xarray (<https://geohackweek.github.io/nDarrays/>), and Abernathy's book: ([https://earth-env-data-science.github.io/lectures/xarray/xarray\\_intro.html](https://earth-env-data-science.github.io/lectures/xarray/xarray_intro.html)).

Think of Xarray as a bit like numpy, except that it has a better 'understanding' of the dimensions of your data.

## Multidimensional Arrays

(not as scary as they sound)

### Like a table in Excel but with more than two dimensions

Let's start by drawing some 2D arrays (familiar spreadsheets) on the board. List a bunch of types of data that might fit in that format.

Now extend this idea: what are common examples of 3D arrays? What type of ocean/earth/atmos data might fit this format? Can we see how they are easily related to 2D arrays?

```
In [1]: # list some types of N-d arrays that you might work with

# A movie is a time series of 2D images, so is a 3D dataset

# A series of satellite images is a bit like a movie too, so is also 3D

# Output from numerical simulation might be 3D or 4D even
```

## What is Xarray?

Xarray is a python library that brings the convenient features of Pandas to higher dimensions. It was initially developed by people at the Climate Corporation for dealing with climate data. It turns out it's very useful in much of our work.

Like Pandas, Xarray lets us do operations on named dimensions. It keeps a lot of the

'metadata' associated with the actual data that makes plotting, calculating, and grouping much easier.

It also is a great way to load data formats that we will see a lot in our work, in particular NetCDFs (more on this later).

features:

the `DataArray` is Xarray's standard data type: a labeled, multi-dimensional array

the `DataArray` has these key properties:

- `data` : N-dimensional array (NumPy or dask) holding the array's values, i.e. your actual data,
- `dims` : dimension names for each axis, just the names, like 'latitude' or 'longitude' or 'time'
- `coords` : dictionary-like container of arrays that label each point, i.e. the actual values of each axis like time or latitude or something
- `attrs` : ordered dictionary holding metadata, or 'attributes', like the data units, person who collected, any of that stuff

The second main data structure is the `Dataset` - which you can think of as a group of `DataArray`, like perhaps a grid that has a temperature `DataArray`, a salinity `DataArray`, and an oxygen `DataArray` that are all from the same instrument or model and belong together. These can be housed within one `Dataset`. We will show this later

## When to use Xarray?

- if your data are multidimensional, like lat, lon, x, y, z, time ...
- if your data are on a regular grid (output from a model)
- if your data are contained in a `.nc` netcdf file.

## What can Xarray do for *me*?

Think about how you would want to build a data structure. Let's go through on the board all the things that should go along with your data. For concreteness let's think about the temperature at all depths and points in the mid-Atlantic Bight, from a model say. What do we need to know? We can name all those vectors, but wouldn't it be easier if they all

lived with the data itself??

## Build a simple DataArray

That is maybe a lot of complex info. Let's start from the basic basics and build up our own `DataArray` using Xarray to see how all it's parts work

We will start with some fake data, just a sequence of numbers. Make up something about what they are.

We need to download xarray using `conda` like we have before:

### installing `xarray` with (Ana)conda (if you don't have it installed already or there was a problem)

to get `xarray` (you should only need to do this once):

1. go to the little `+` on the left to open a `launcher` window.
2. click the 'terminal' tile to open a terminal
3. type `conda env list` and hit enter. Make sure that there is a `*` next to the line that says swbc
4. if that's right type `conda install xarray dask netCDF4 bottleneck pandas`
5. when it asks `proceed ([y]/n)?` type `y` and hit enter

Of course, to start we need to import xarray. To save typing we usually ask python to call xarray 'xr'

```
In [2]: # import xarray and matplotlib
import xarray as xr
import matplotlib.pyplot as plt
%matplotlib inline
```


Now let's use the `xr.DataArray()` function to make our fake dataarray

```
In [3]: da = xr.DataArray([ 1, 2, 5, 7, 10])

da
```

Out [3]: xarray.DataArray (dim\_0: 5)

---

 array([ 1, 2, 5, 7, 10])

► Coordinates: (0)

► Indexes: (0)

► Attributes: (0)


Ok, that is just some data, a few numbers. So far this isn't anything more special than a numpy array. But it'll get better if we can give it some dimensions and coordinates that give the data context. Maybe those are the number of birds we counted at 10 minute intervals while we were staring out the window in class.

Let's add the dimension `time` to the DataArray:

```
In [4]: da = xr.DataArray([ 1, 2, 5, 7, 10], dims=["time"])
da
```

Out [4]: xarray.DataArray (time: 5)

---

 array([ 1, 2, 5, 7, 10])

► Coordinates: (0)

► Indexes: (0)

► Attributes: (0)

Ok, now we know that the data correspond to times, that's good. It's giving us a more complete understanding of the context of the data. But what times?

We've named the dimension, but now let's give it some coordinates, in otherwords let's specify the exact times when we measured those birds.

In order to connect the dimension to an actual set of time points (i.e. coordinates) we need to recall a Python data type we talked about a while ago.

## Quick Aside: Dictionaries - remember that python data type?

Quick summary: A Dictionary is a lot like a list, but it has a label for each element in the list. Namely a Dictionary is made up of key : values pairs. You can think of it literally like a dictionary: you'd look something by it's name (key) and there would be other information in there (the values). We define a dictionary using curly brackets: `dict = {}` and we separate keys from their values with a colon `:`

```
d = {
    <key>: <value>,
    <key>: <value>,
    .
    .
    .
    <key>: <value>
}
```

For example we can make a dictionary where the key is a city, and the value is that city's baseball team:

```
MLB_team = {
    'Colorado' : 'Rockies',
    'Boston'    : 'Red Sox',
    'Minnesota' : 'Twins',
    'Milwaukee' : 'Brewers',
    'Seattle'   : 'Mariners'
}
```

Dictionaries aren't too complicated, they are just another way of storing data. We will use them a lot with xarrays.

One use case we can talk about now, we have decided the dimension of our DataArray is time, and we can use a dictionary to associate that key (time) with values ( t1, t2, t3, etc).


We've named the dimension, but now let's give it some coordinates, in otherwords let's specify the exact times when we measured those birds. Just to be simple we will say starting at 0 minutes we looked out every 10 minutes to count the birds.

When we specify the coordinates, we need to tell xarray that we are referring to the `time` dimension, and then tell it what times we want to include:

```
In [5]: da = xr.DataArray( [ 1, 2, 5, 7, 3],
                        dims=['time'],
                        coords = {'time': range(5)})

# da['time'] = [1,2,3,4,5]
da
```

```
Out [5]: xarray.DataArray    (time: 5)
```

 array([1, 2, 5, 7, 3])

▼ Coordinates:

| time | (time) | int64 | 0 | 1 | 2 | 3 | 4 |
|------|--------|-------|---|---|---|---|---|
|      |        |       |   |   |   |   |   |




► Indexes: (1)

► Attributes: (0)

```
In [6]: # da.attrs = {'data':'bird observations','time':'minutes'}
da.attrs = {'units':'bird observations','time':'minutes'}
da
```

```
Out[6]: xarray.DataArray (time: 5)
```

 array([1, 2, 5, 7, 3])

▼ Coordinates:

| time  | (time) | int64 | 0 | 1 | 2 | 3 | 4 |
|---|--------|-------|---|---|---|---|---|
|   |        |       |   |   |   |   |   |

► Indexes: (1)

▼ Attributes:

|         |                   |
|---------|-------------------|
| units : | bird observations |
| time :  | minutes           |

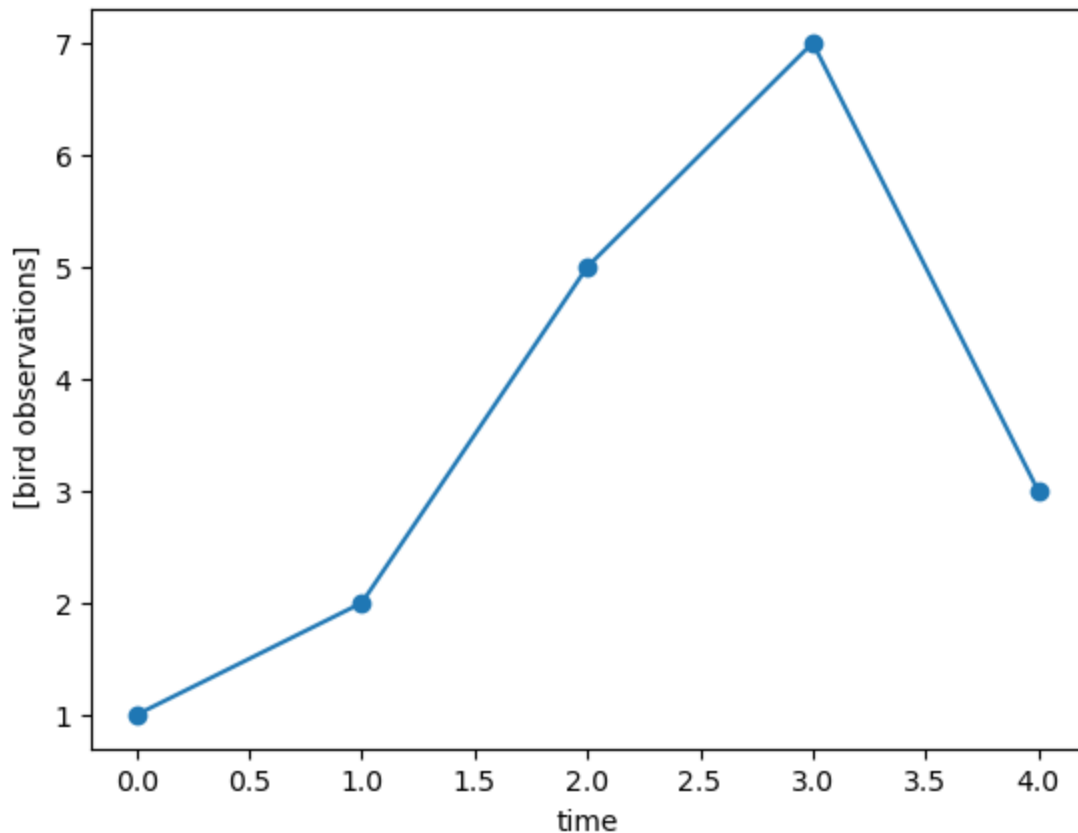
Now we have something that pretty much fully describes our experiment (almost, what might we be missing? what about attribute? can you think of some examples?)

From here Xarray has all sorts of built in functions that allow you to interact with your data quickly. Because the DataArray has data, dimensions, and coordinates, Xarray can internally understand a lot about the data.

One thing we can do is just ask xarray to make a simple plot. This takes almost no code:

```
In [7]: # da.plot()
da.plot(marker='o')
da.plot
# or do shifttab after da.plot to see help
```

```
Out[7]: <xarray.plot.accessor.DataArrayPlotAccessor at 0x17a57ed40>
```



The end...

## Breakout / exercise 01

### Build a multidimensional DataArray

We made up some data in the simple example. Also, did you notice it's just one dimensional? Let's go through the exercise by building some a multidimensional dataset.

This is in the next notebook

### Exercise 01 - Make your own DataArray

For this exercise we are going to steal a function from the library `pandas`, even though we didn't learn about it yet. The function is called `date_range` and it creates an array of dates. Look at the code below, can you figure out what it does?

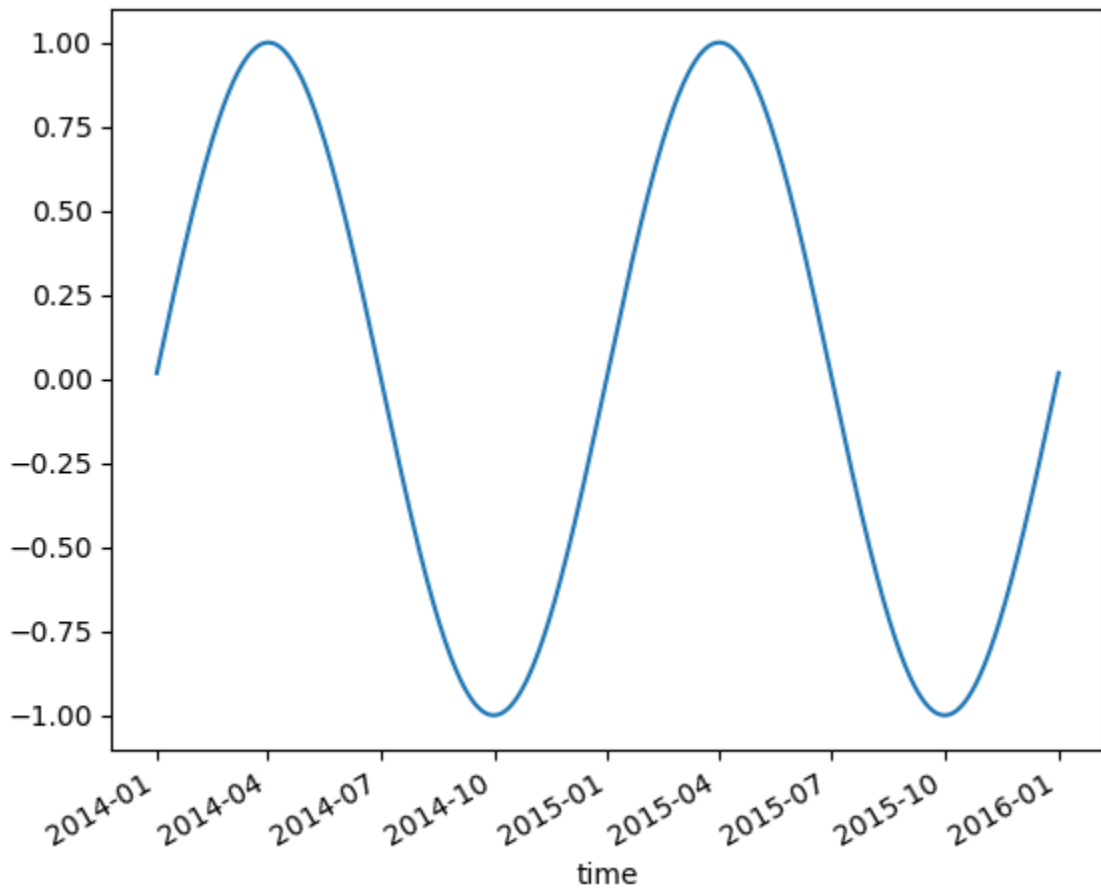
**Task:** Make an xarray dataarray of fake data. Make a dim for time, for the coordinates use `two_years` and for the data use `fake_data` defined as:

```
import pandas as pd
```

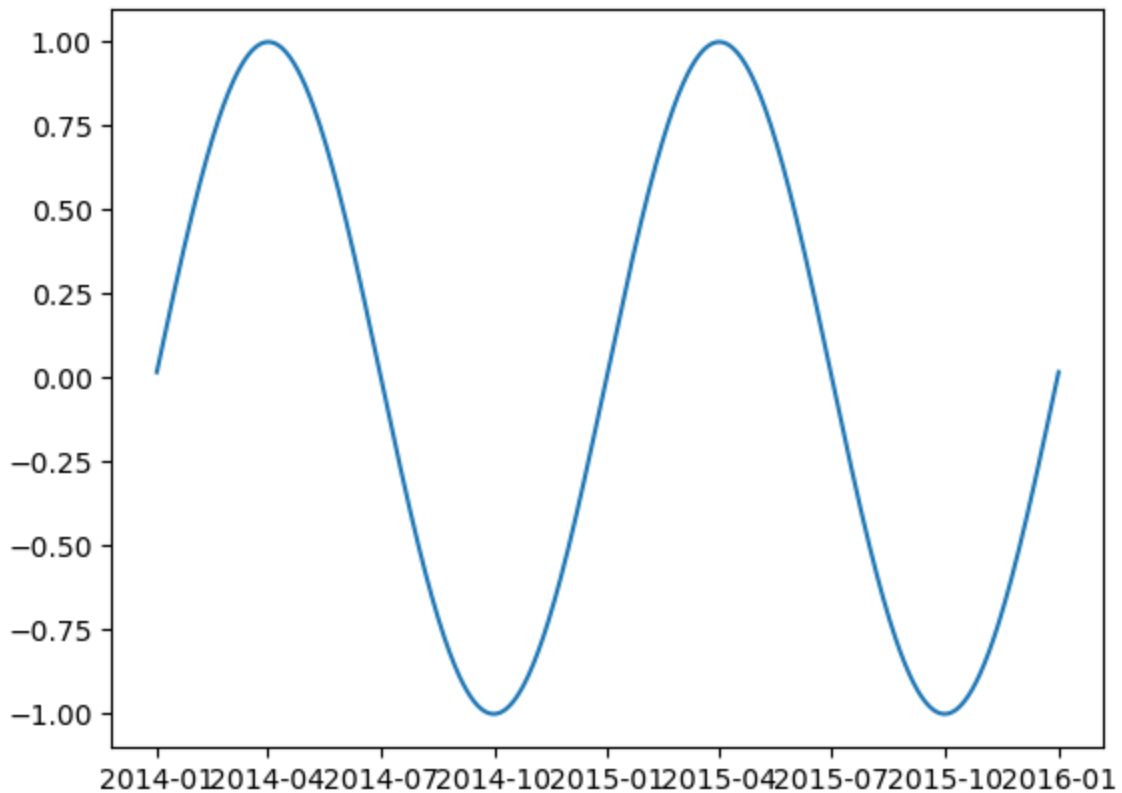
```
two_years = pd.date_range(start='2014-01-01', end='2016-01-01',  
freq='D')  
fake_data = np.sin(2 * np.pi * two_years.dayofyear / 365)  
Make a default plot of your dataarray and print out the data contents
```

Don't forget to import xarray and numpy!

```
In [81]: import pandas as pd  
import numpy as np  
import xarray as xr  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
two_years = pd.date_range(start='2014-01-01', end='2016-01-01', freq='D')  
fake_data = np.sin(2 * np.pi * two_years.dayofyear / 365)  
  
da = xr.DataArray( fake_data,  
                    dims=['time'],  
                    coords = {'time': two_years})  
da.plot()  
  
fig = plt.figure();  
plt.plot(two_years, fake_data);  
da;
```







## Build a multidimensional DataArray and Dataset

We made up some data in the simple example. Also, did you notice it's just one dimensional? Let's go through the exercise by building a multidimensional dataset.

We are going to start with some data that is just a bunch of normal numpy arrays, so we need to load numpy as well as xarray

### credit

This lesson is from Abernathy's book: ([https://earth-env-data-science.github.io/lectures/xarray/xarray\\_intro.html](https://earth-env-data-science.github.io/lectures/xarray/xarray_intro.html)).

```
In [9]: # import xarray, numpy and matplotlib
import xarray as xr
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

## Example: ARGO float

Let's start by loading some real data. This is ARGO float data that contains temperature and salinity data {What is an Argo float? how does it take data?}. Those data are in the form of numpy arrays, or matrices. So, again, rows and columns. Let's draw on the board what the rows and columns are. They have coordinates like time, depth, latitude, longitude. Stuff you would expect to describe data collected in the ocean.

Right now, when we load the data, it's going to be a collection of numpy arrays. They are all separate objects, and what we'd like to do is stitch them together in a sensible way. To this we are going to create a DataArray, then a Dataset.

```
In [10]: argo_data = np.load('../data/argo_float_4901412.npz')
list(argo_data.keys())
# dict(argo_data)
# type conversion... int("1"), str(1)
```

```
Out[10]: ['S', 'T', 'levels', 'lon', 'date', 'P', 'lat']
```

They are in this container because of how they are saved. Let's break each component out into its own numpy array

```
In [11]: S = argo_data['S']
T = argo_data['T']
P = argo_data['P']
levels = argo_data['levels']
lon = argo_data['lon']
lat = argo_data['lat']
date = argo_data['date']

# %whos

for key in argo_data:
    print( key, ' shape is: ', argo_data[key].shape)
```

```
S shape is: (78, 75)
T shape is: (78, 75)
levels shape is: (78,)
lon shape is: (75,)
date shape is: (75,)
P shape is: (78, 75)
lat shape is: (75,)
```

Remember from the previous notebook.

The `DataArray` has these key properties:

- `data` : N-dimensional array (NumPy or dask) holding the array's values, i.e. your actual data,
- `dims` : dimension names for each axis, just the names, like 'latitude' or 'longitude' or 'time'

- `coords` : dictionary-like container of arrays that label each point, i.e. the actual values of each axis like time or latitude or something
- `attrs` : ordered dictionary holding metadata, or 'attributes', like the data units, person who collected, any of that stuff

Let's take the salinity `S` and create a `DataArray` for it

```
In [12]: da_salinity = xr.DataArray(S, dims=['level', 'date'], coords={'level': level
# da_salinity = xr.DataArray(S, coords={'level': levels, 'date': date})
# da_salinity = xr.DataArray(S, dims=['level', 'date'])

da_salinity
```

```
Out[12]: xarray.DataArray    (level: 78, date: 75)
```

```
array([[35.6389389 , 35.51495743, 35.57297134, ..., 35.82093811,
        35.77793884, 35.66891098],
       [35.63393784, 35.5219574 , 35.57397079, ..., 35.81093216,
        35.58389664, 35.66791153],
       [35.6819458 , 35.52595901, 35.57297134, ..., 35.79592896,
        35.66290665, 35.66591263],
       ...,
       [34.91585922, 34.92390442, 34.92390442, ..., 34.93481064,
        34.94081116, 34.94680786],
       [34.91585922, 34.92390442, 34.92190552, ..., 34.93280792,
        34.93680954, 34.94380951],
       [34.91785812, 34.92390442, 34.92390442, ...,      nan,
        34.93680954,      nan]])
```

▼ Coordinates:

| level | (level) | int64          | 0                          | 1   | 2 | 3 | 4 | 5 | 6 | ... | 72 | 73 | 74 | 75 | 76 | 77 |  |  |
|-------|---------|----------------|----------------------------|-----|---|---|---|---|---|-----|----|----|----|----|----|----|--|--|
| date  | (date)  | datetime64[ns] | 2012-07-13T22:33:06.019200 | ... |   |   |   |   |   |     |    |    |    |    |    |    |  |  |

► Indexes: (2)

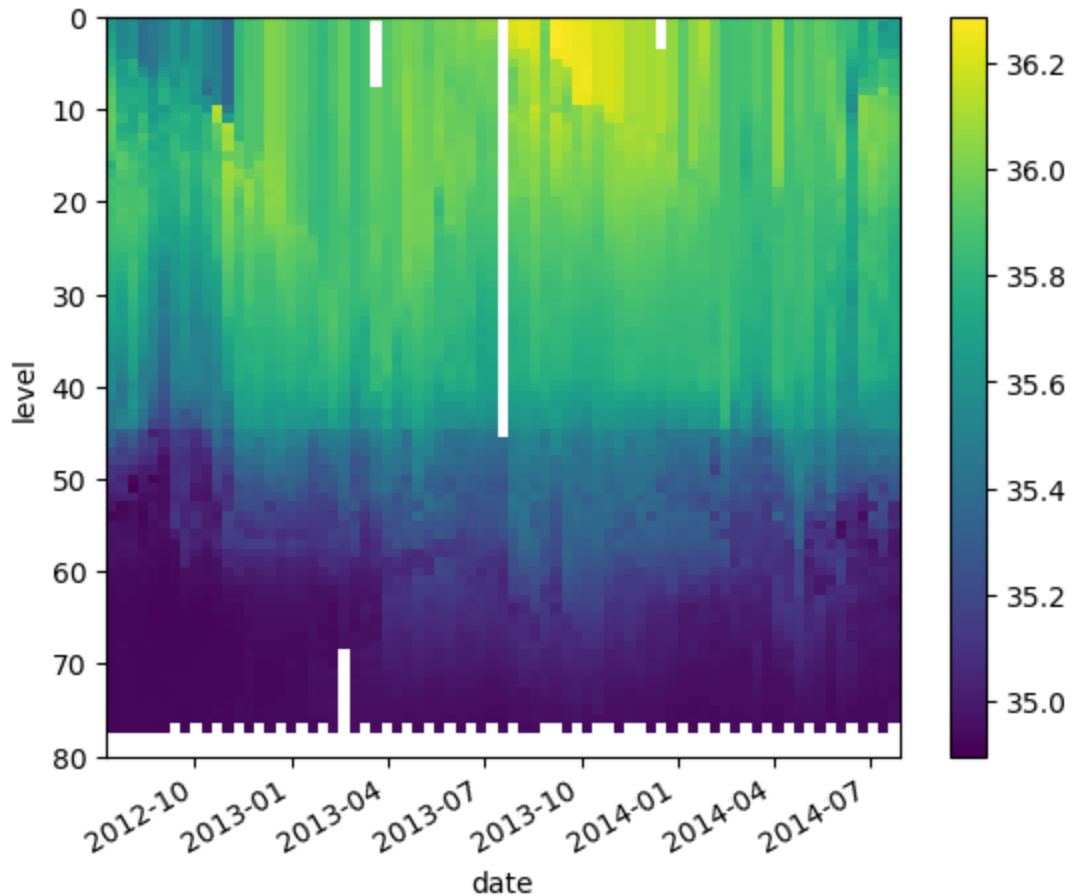
► Attributes: (0)

Ok, this is like the 1D fake bird data we made before, but now it's real 2D salinity data from the ocean.

Let's see what `xarray` does if we ask it to make a simple plot:

```
In [13]: da_salinity.plot()

# lets switch the axis order so depth goes down
plt.ylim([80, 0]);
```



Nice! That is sort of amazing. Xarray knew that the salinity data is 2d - so by default it smartly made a pcolor plot (not a line plot or something). It also knew that time is on the x axis, and the 'levels' (depth) are on the y axis because the dimensions match. It also labeled out axis and formatted the dates.





But we aren't done with our DataArray yet. Remember the four parts of a DataArray? `data`, `dims`, `coords`, `attrs`. We can add other important information into the `attrs` part of the DataArray. Can you think of some important info?

```
In [14]: # add some attributes describing the data and units
da_salinity.attrs['units'] = 'PSU'
da_salinity.attrs['standard_name'] = 'sea_water_salinity'
da_salinity
```

Out[14]: xarray.DataArray (level: 78, date: 75)

```
array([[35.6389389, 35.51495743, 35.57297134, ..., 35.82093811,
        35.77793884, 35.66891098],
       [35.63393784, 35.5219574, 35.57397079, ..., 35.81093216,
        35.58389664, 35.66791153],
       [35.6819458, 35.52595901, 35.57297134, ..., 35.79592896,
        35.66290665, 35.66591263],
       ...,
       [34.91585922, 34.92390442, 34.92390442, ..., 34.93481064,
        34.94081116, 34.94680786],
       [34.91585922, 34.92390442, 34.92190552, ..., 34.93280792,
        34.93680954, 34.94380951],
       [34.91785812, 34.92390442, 34.92390442, ..., nan,
        34.93680954, nan]])
```

▼ Coordinates:

|              |         |                |                                 |   |   |   |   |   |   |     |    |    |    |    |    |    |   |   |
|--------------|---------|----------------|---------------------------------|---|---|---|---|---|---|-----|----|----|----|----|----|----|---|---|
| <b>level</b> | (level) | int64          | 0                               | 1 | 2 | 3 | 4 | 5 | 6 | ... | 72 | 73 | 74 | 75 | 76 | 77 |  |  |
| <b>date</b>  | (date)  | datetime64[ns] | 2012-07-13T22:33:06.019200 .... |   |   |   |   |   |   |     |    |    |    |    |    |    |  |  |

► Indexes: (2)

▼ Attributes:

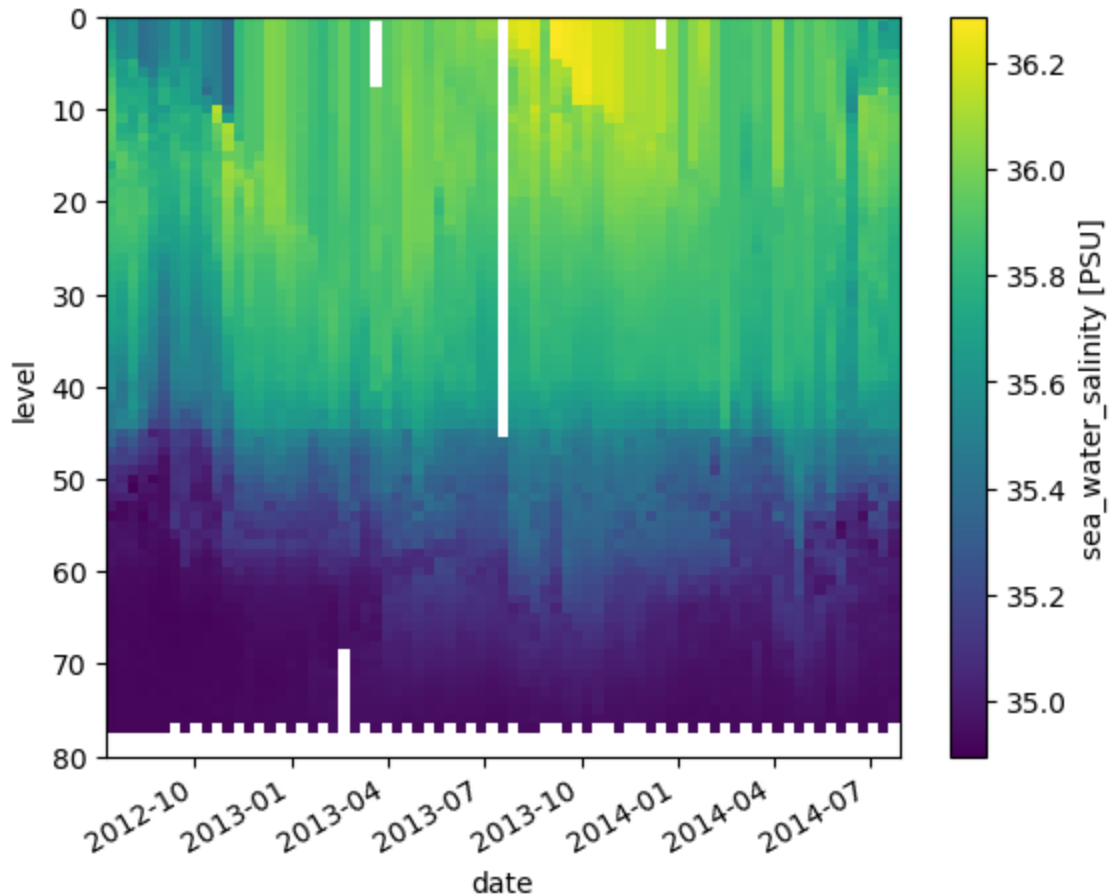
units : PSU  
standard\_name : sea\_water\_salinity

In [15]: *# make the default plot again:*

```
da_salinity.plot()

plt.ylim([80, 0])
```

Out[15]: (80.0, 0.0)



## Datasets

xarray datasets can hold multiple DataArrays. This makes particular sense if the data in those multiple DataArrays share dimensions and coordinates.

In our ARGO float example, both the Temperature and Salinity share the same dims and coords. So let's put them together into one dataset that holds all out float observational data.

The Dataset constructor takes three arguments:

- `data_vars` should be a dictionary with each key as the name of the variable and each can be an already constructed DataArray, or a tuple that looks like this `(dims, data[, attrs])`
- `coords` should be a dictionary of the same form as `data_vars`.
- `attrs` should be a dictionary.

So here is an example for our argo data:

```
In [16]: # A dictionary of data variables that specifies the dimensions and data as a
data_vars = {'salinity': (('level', 'date'), S),
```

```

        'temperature': (('level', 'date'), T),
        'pressure':    (('level', 'date'), P)}

# A dictionary of coordinates
coords={'level': levels, 'date': date}

argo = xr.Dataset(data_vars, coords)





# print
argo

```

Out[16]: xarray.Dataset

► Dimensions: (level: 78, date: 75)

▼ Coordinates:

|              |         |                |                                 |   |
|--------------|---------|----------------|---------------------------------|---|
| <b>level</b> | (level) | int64          | 0 1 2 3 4 5 6 ... 72 73 74 7... |   |
| <b>date</b>  | (date)  | datetime64[ns] | 2012-07-13T22:33:06.019...      |   |

▼ Data variables:

|                    |               |         |                                 |   |
|--------------------|---------------|---------|---------------------------------|---|
| <b>salinity</b>    | (level, date) | float64 | 35.64 35.51 35.57 ... nan ...   |   |
| <b>temperature</b> | (level, date) | float64 | 18.97 18.44 19.1 ... nan 3.7... |   |
| <b>pressure</b>    | (level, date) | float64 | 6.8 6.1 6.5 5.0 ... nan 2e+...  |   |

► Indexes: (2)

► Attributes: (0)

Let's talk through what all those parts are telling us when we print out `argo`.

What about the latitude and longitude? Those seem important and we'd like to use them for plotting and analysis later. They should be coordinates right? They should be the same size as one of the existing coordinates, either level or date. what do you think?

to add a new coordinate we can use:

```

In [17]: argo.coords['lon'] = lon # or argo_data['lon']

argo

del argo['lon'];

```

What we just did was add a whole new coordinate `lon`. But actually we know that each `lon` point is at a particular `date` location. So actually we can associate `lon` and `date`. To do that we set the dim of our new coord `lon` to be `date`. We can do the same for `lat`.

```









In [18]: argo.coords['lon'] = ('date', lon)
argo.coords['lat'] = ('date', lat)
argo

```







Out[18]: xarray.Dataset

► Dimensions: (level: 78, date: 75)

▼ Coordinates:

|       |         |                |                                 |   |
|-------|---------|----------------|---------------------------------|---|
| level | (level) | int64          | 0 1 2 3 4 5 6 ... 72 73 74 7... |   |
| date  | (date)  | datetime64[ns] | 2012-07-13T22:33:06.019...      |   |
| lon   | (date)  | float64        | -39.13 -37.28 ... -34.11 -3...  |   |
| lat   | (date)  | float64        | 47.19 46.72 46.45 ... 42.4...   |   |

▼ Data variables:

|             |               |         |                                 |   |
|-------------|---------------|---------|---------------------------------|---|
| salinity    | (level, date) | float64 | 35.64 35.51 35.57 ... nan ...   |   |
| temperature | (level, date) | float64 | 18.97 18.44 19.1 ... nan 3.7... |   |
| pressure    | (level, date) | float64 | 6.8 6.1 6.5 5.0 ... nan 2e+...  |   |

► Indexes: (2)

► Attributes: (0)

## Working with labeled data

We've built our nice dataset for the ARGO float. It has Temperature, salinity and pressure data. Those data also have label dimensions / coordinates that include level, date, lat, and lon.

Now we are going to start to see some of the power of Xarray and how those labeled dimensions / coordinates let us make our analysis easier

## Selecting data (indexing)

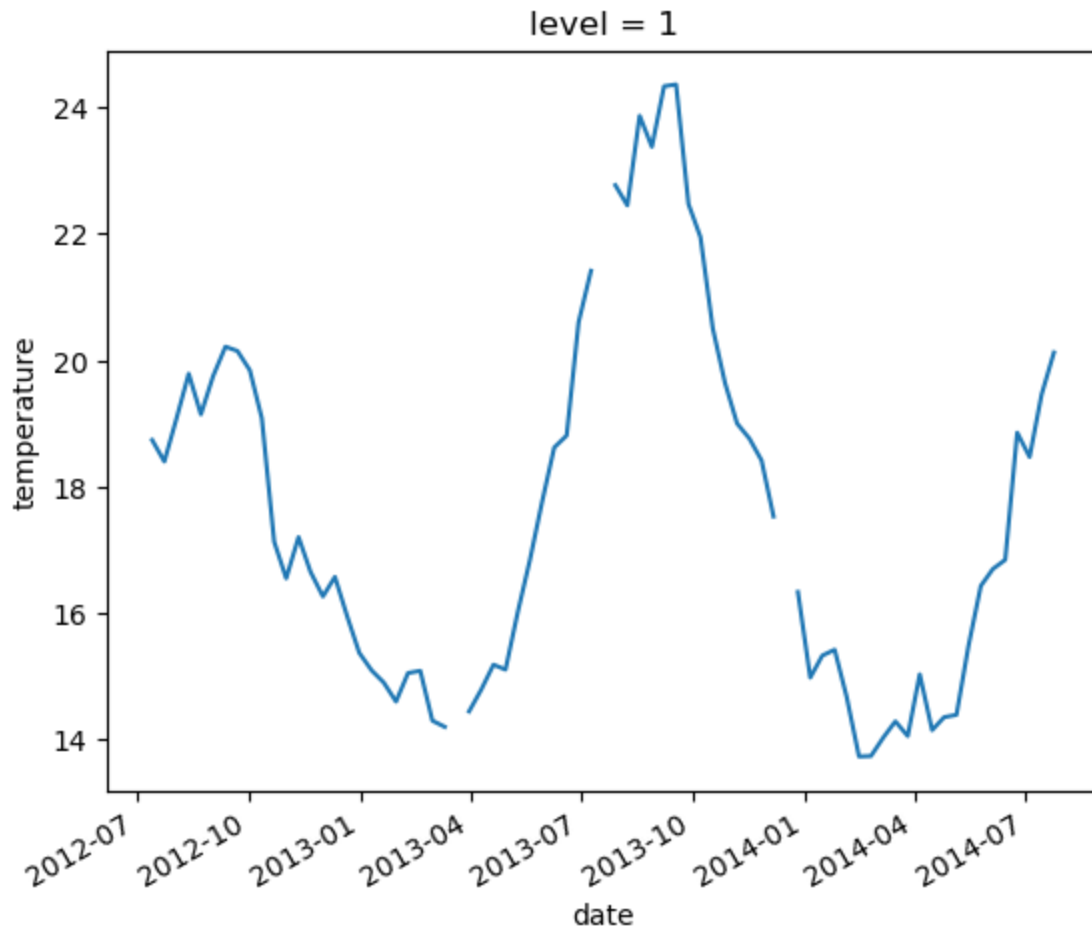
Let's say we want to look at some subset of the temperature data (just a slice). We can use standard numpy notation to do this by indicating the number of the row and column we are interested in. Let's say we want to look at the second row and all columns ( so this is like a timeseries at a particular level)

using standard numpy indexing this would look like:

```
In [19]: # plot temp at level 1 over time:
argo.temperature[1,:].plot()
# argo.temperature.sel(levels = 5).plot()
```

Out[19]: [<matplotlib.lines.Line2D at 0x17a6d7a90>]



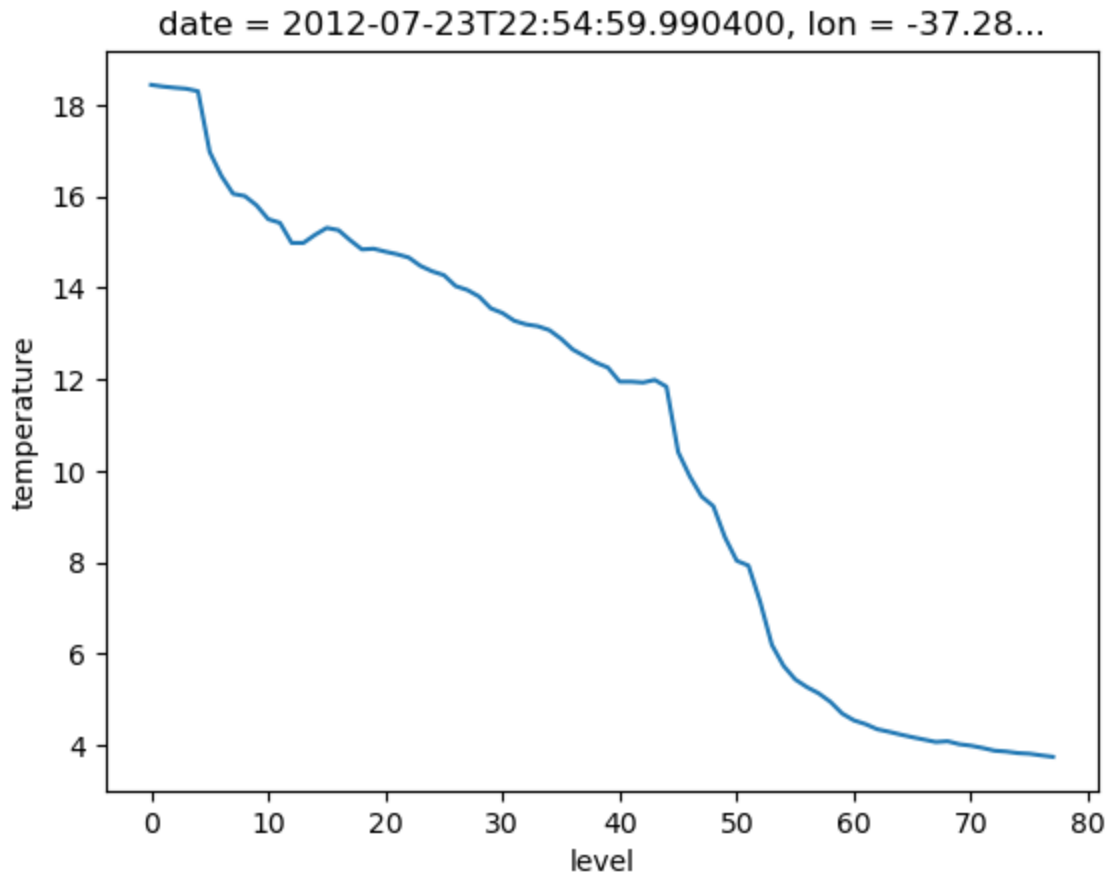


what about a particular depth profile?

in standard numpy indexing:

```
In [20]: # plot temp vs level for profile 1  
argo.temperature[:,1].plot()
```

```
Out[20]: [<matplotlib.lines.Line2D at 0x17a939350>]
```



That seems easy enough. But let's say you want to look at the temperature profile from a particular day. How are you going to do that? Well, you'd need to use the `date` dimension, look up the date you want, find it's index (meaning the number/position it comes in the list of dates) then put that index into the `argo.temperature[:,1].plot()` line.

This isn't impossible. This is the kind of thing you do all the time in matlab. It's annoying and takes a few lines of code.

But xarray solves this problem! using the `.sel()` method you can 'select' a part of your data based on the label.

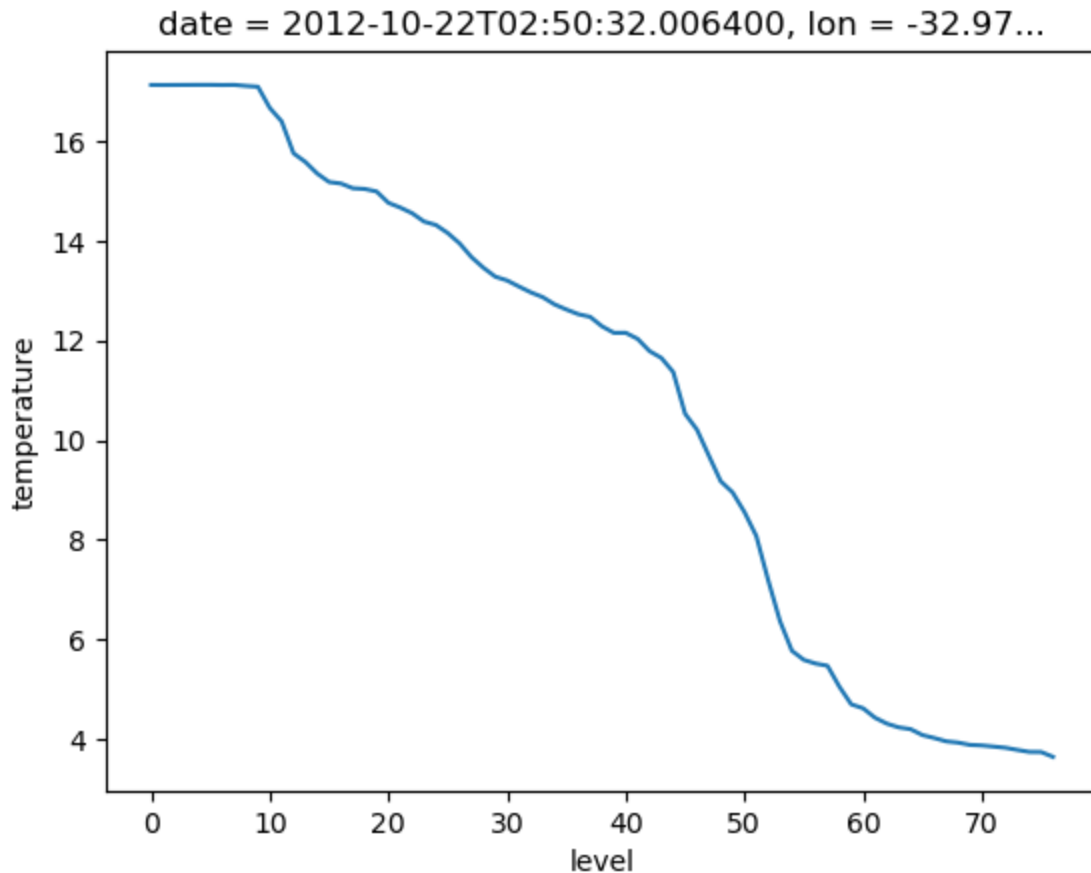
Here is how it works. let's get the profile on Oct 22 2012 by selecting based on the dimension `date` :

```
In [21]: # plot the temp profile on a particular day using .sel()

argo.temperature.sel(date='10-22-2012', method='nearest').plot()
# argo.temperature.sel(date='10-23-2012', method='nearest').plot()

# argo.temperature.sel(date='10-22-2012').plot(y='level')
```

```
Out[21]: [<matplotlib.lines.Line2D at 0x17aa636d0>]
```



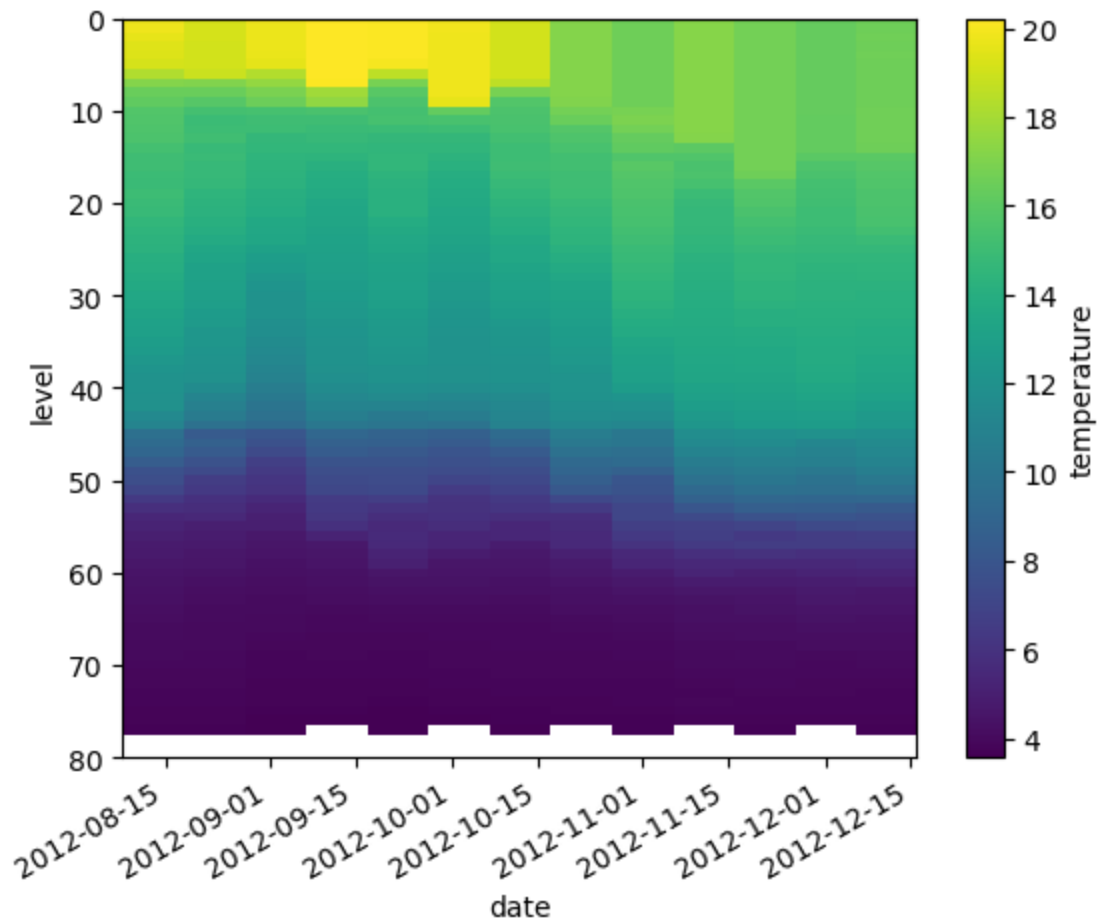
## Slicing data

We can also grab a bunch of days. Grabbing a bunch of consecutive data is typically called 'slicing'. We have to tell xarray that we want a slice of the `date` dimension. Again, this is new syntax, so don't be worried that you don't know it. You'll learn as you go from examples and from reading the documents for different packages.

let's get a couple months around our previous profile:

```
In [22]: # select a time range using .sel() with the slice() argument
argo.temperature.sel( date = slice('08-10-2012','12-10-2012') ).plot()
plt.ylim([80,0])
# plt.gcf().autofmt_xdate()
```

```
Out[22]: (80.0, 0.0)
```



You can also use `.sel()` on the whole dataset to, for example, grab all your data from one day:









```
In [23]: argo_one_day = argo.sel(date='10-22-2012')
# argo_one_day = argo.sel(lon='-37.053') # doesn't work

argo_one_day
```







Out [23]: xarray.Dataset

► Dimensions: (level: 78, date: 1)

▼ Coordinates:

|       |         |                |                                 |   |
|-------|---------|----------------|---------------------------------|---|
| level | (level) | int64          | 0 1 2 3 4 5 6 ... 72 73 74 7... |   |
| date  | (date)  | datetime64[ns] | 2012-10-22T02:50:32.00...       |   |
| lon   | (date)  | float64        | -32.97                          |   |
| lat   | (date)  | float64        | 44.13                           |   |

▼ Data variables:

|             |               |         |                                 |   |
|-------------|---------------|---------|---------------------------------|---|
| salinity    | (level, date) | float64 | 35.47 35.47 35.47 ... 34.9...   |   |
| temperature | (level, date) | float64 | 17.13 17.13 17.13 ... 3.639 ... |   |
| pressure    | (level, date) | float64 | 6.4 10.3 15.4 ... 1.951e+03...  |   |

► Indexes: (2)

► Attributes: (0)

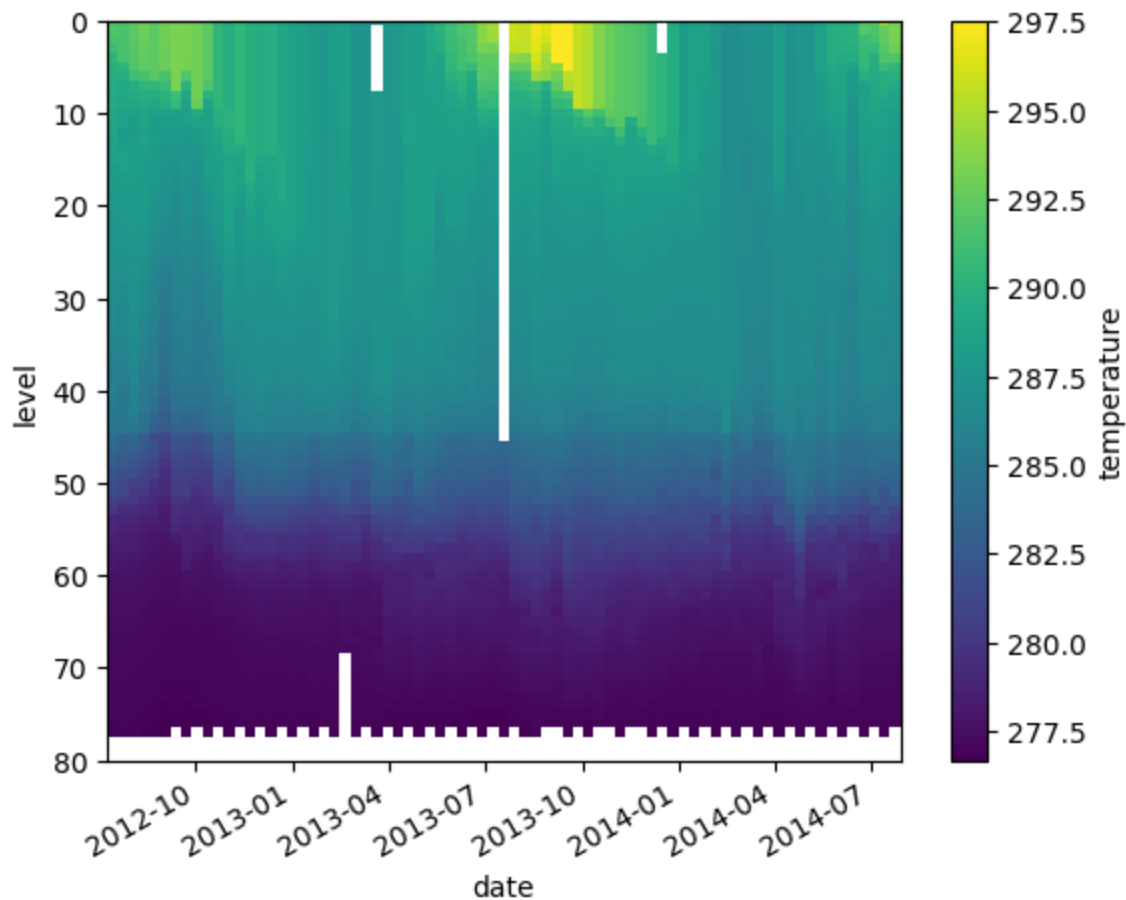
## Math

we can do any normal math on these DataArrays and Datasets:

```
In [24]: temp_kelvin = argo.temperature + 273.15

temp_kelvin.plot()
plt.ylim([80,0])

argo['temp_kelvin'] = temp_kelvin;
# argo
```



you can combine DataArrays of the same size to get derived products like buoyancy:

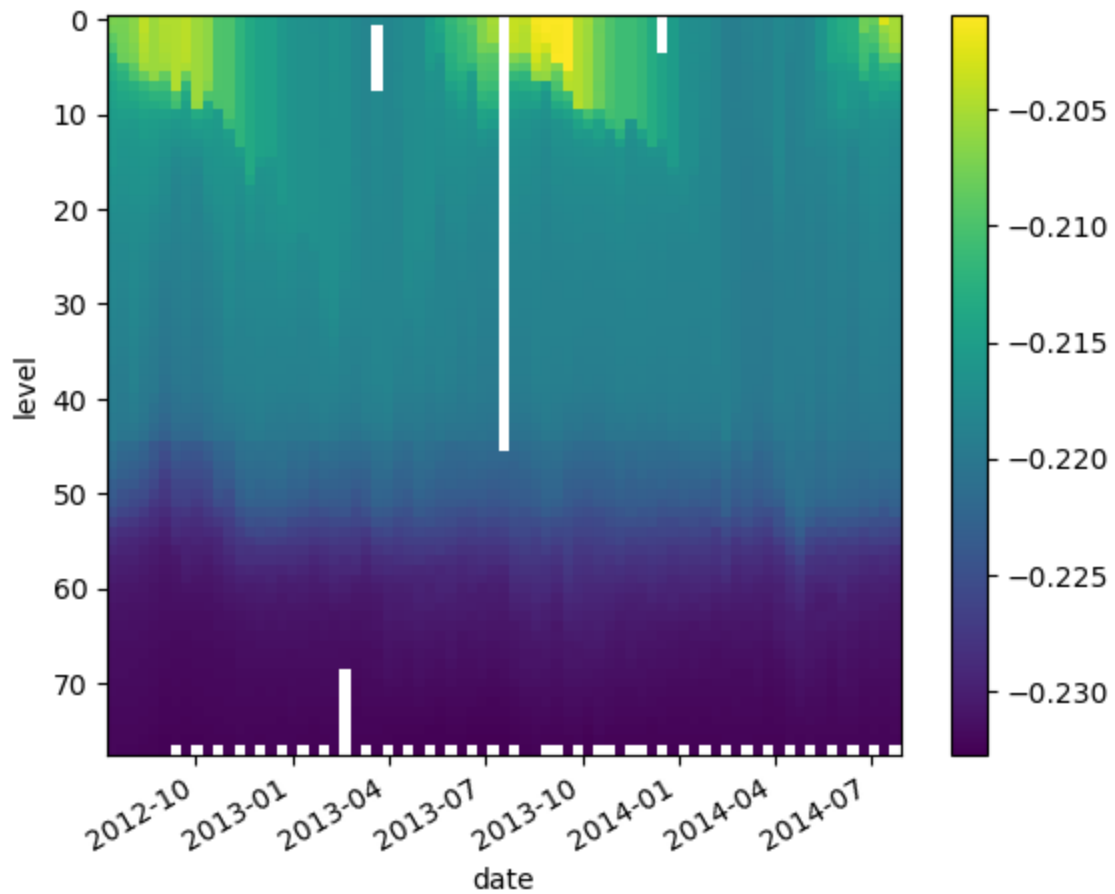
```
In [25]: g = 9.8
alpha = 2e-4
beta = -7e-4

buoyancy = g * (alpha * argo.temperature + beta * argo.salinity)

buoyancy.plot()

# plt.ylim([80,0])
plt.gca().invert_yaxis()

argo['buoyancy'] = g * (alpha * argo.temperature + beta * argo.salinity);
# argo
```



we can do standard numpy math stuff like means, standard deviations, etc on dimensions.

We can average the whole dataset. xarray is smart, and it's going to average each of the data variables independantly:

```
In [26]: argo_mean = argo.mean(dim='date')  
  
argo_mean
```






Out [26]: xarray.Dataset

► Dimensions: (level: 78)

▼ Coordinates:

|       |         |       |                                     |   |
|-------|---------|-------|-------------------------------------|---|
| level | (level) | int64 | 0 1 2 3 4 5 6 ... 72 73 74 75 76 77 |   |
|-------|---------|-------|-------------------------------------|---|

▼ Data variables:

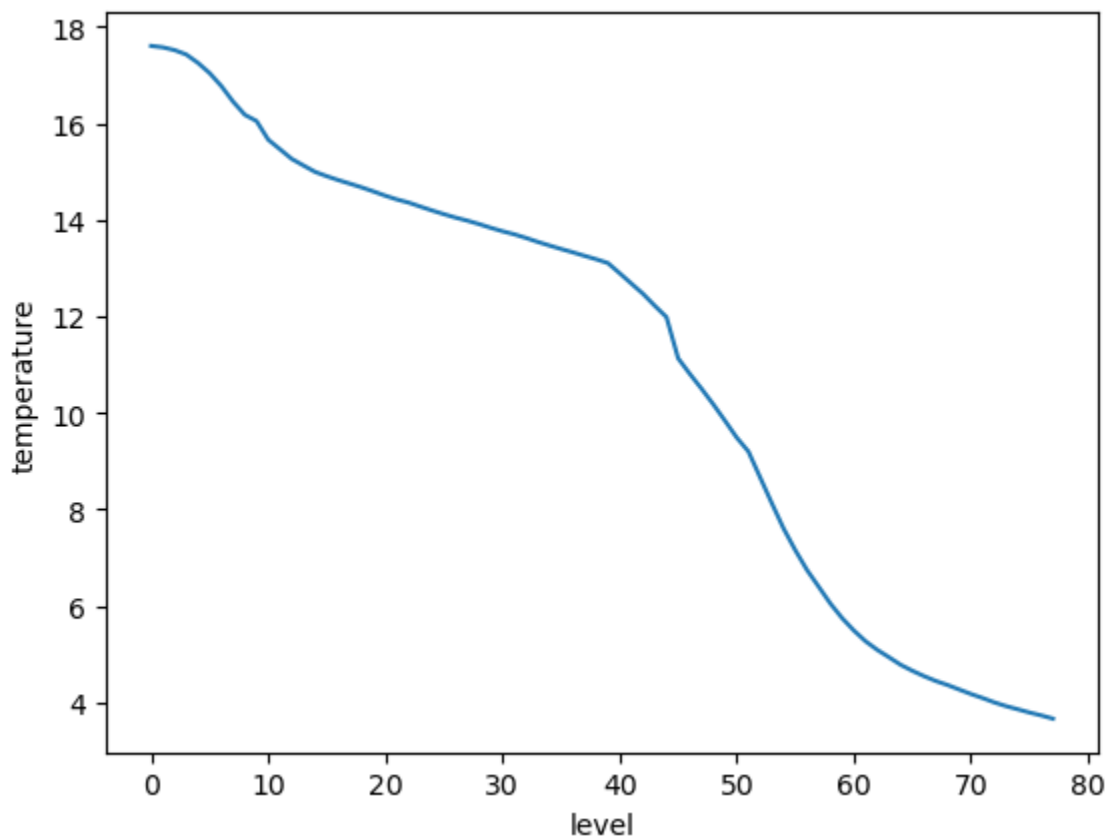
|             |         |         |                                     |   |
|-------------|---------|---------|-------------------------------------|---|
| salinity    | (level) | float64 | 35.91 35.9 35.9 ... 34.94 34.93     |   |
| temperature | (level) | float64 | 17.6 17.57 17.51 ... 3.73 3.662     |   |
| pressure    | (level) | float64 | 6.435 10.57 ... 1.95e+03 1.999e+03  |   |
| temp_kelvin | (level) | float64 | 290.8 290.7 290.7 ... 276.9 276.8   |   |
| buoyancy    | (level) | float64 | -0.2118 -0.2118 ... -0.2324 -0.2325 |   |

► Indexes: (1)

► Attributes: (0)

```
In [27]: argo_mean.temperature.plot()  
# plt.plot(argo_mean.pressure, argo_mean.temperature)
```

Out [27]: [matplotlib.lines.Line2D at 0x17a937a90>]

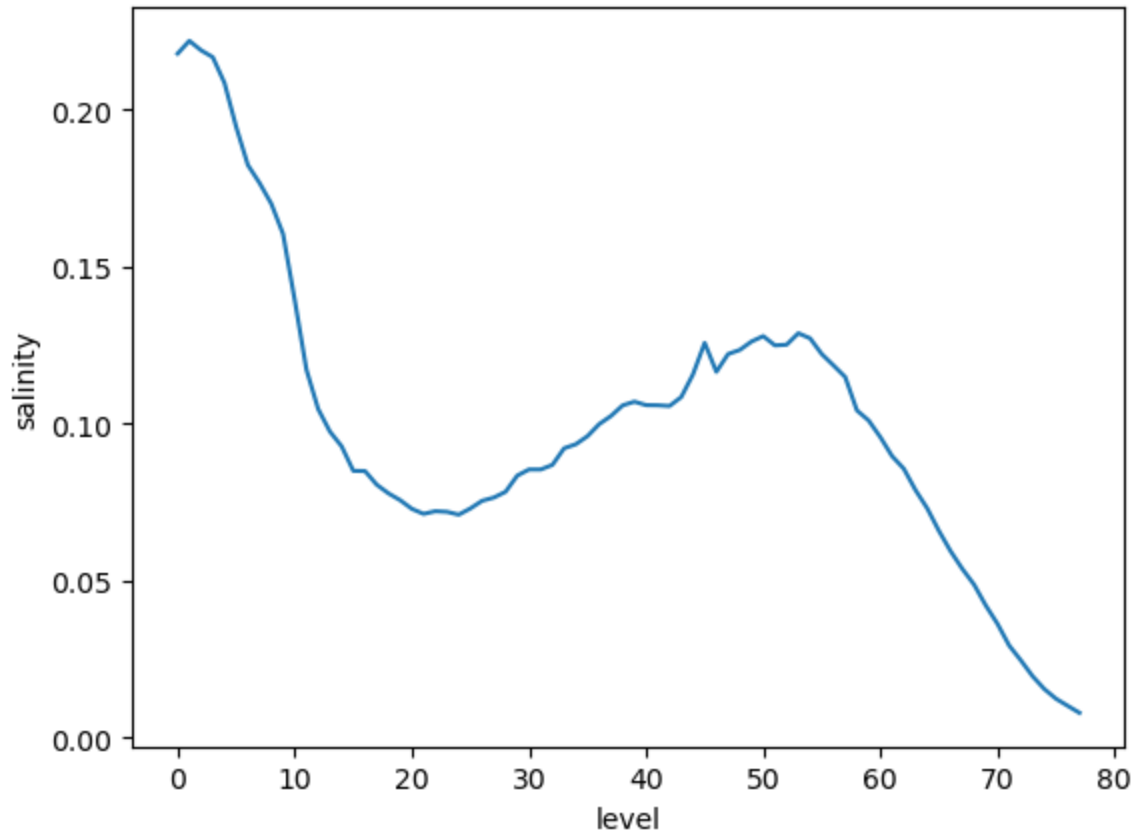


```
In [28]: # get the standard deviation:  
argo_std = argo.std(dim='date')
```



```
# argo_std.temperature.plot()  
argo_std.salinity.plot()
```

Out[28]: [



There are a lot more cool math/analysis functions we can do with xarray. We will see more of them later on.

The end...

## Breakout / exercise 02

### Exercise - plotting and stats

First load what we need and create a dataset (the code below is exactly what we did in the class, compressed into one cell).

```
In [29]: import xarray as xr  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
argo_data = np.load('../data/argo_float_4901412.npz')
```

```

data_vars = {'salinity': (('level', 'date'), argo_data["S"]),
             'temperature': (('level', 'date'), argo_data["T"]),
             'pressure': (('level', 'date'), argo_data["P"])}

# A dictionary of coordinates
coords = {'level': argo_data["levels"], 'date': argo_data["date"]}

argo = xr.Dataset(data_vars, coords)

```

In the lesson, we took the standard deviation of every variable in a DataSet and plotted them like this:

```
argo_std = argo.std(dim='date')
```

```
argo_std.temperature.plot()
```

However, in python, functions can sometimes be chained together. For example:

```
import numpy as np
```

```

x = np.random.rand(10, 10) # Create an array of random numbers
x.mean(axis=1).min() # Compute the mean along one axis, then the
minimum.

```

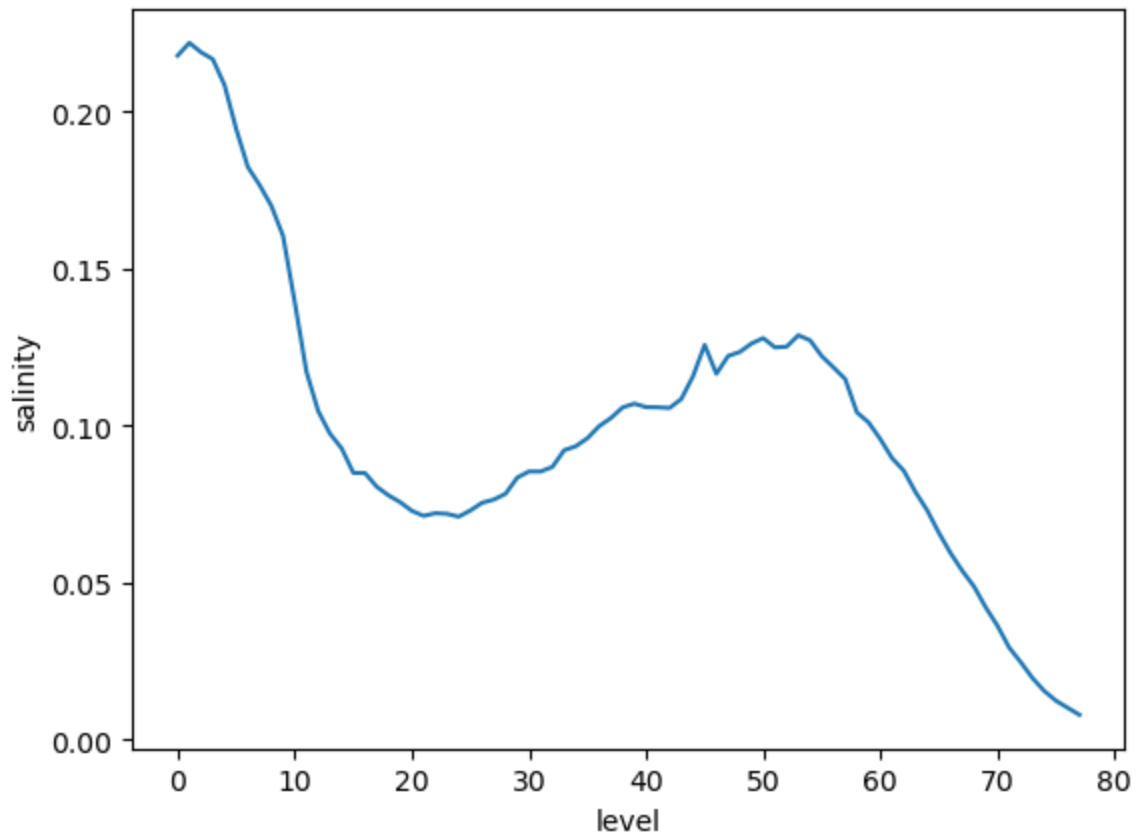
So do we need to create the variable `argo_std` before plotting? Could we do the standard deviation and plot in one line of code? Have a go...

```

In [30]: # argo.salinity.plot()
         argo.salinity.std(dim='date').plot()

```

```
Out[30]: [<matplotlib.lines.Line2D at 0x17ac163d0>]
```



How do you find the maximum standard deviation in salinity? Again, try in one line of code. Does the answer match your expectation from the plot above?

```
In [31]: argo.salinity.std(dim='date').max()
```

```
Out[31]: xarray.DataArray 'salinity'
```

```
array(0.22181831)
```

```
► Coordinates: (0)
```

```
► Indexes: (0)
```

```
► Attributes: (0)
```

## The Network Common Data Format: netCDF

NetCDF is one of the most common ways that geoscience data is distributed. It was developed in the early 1990s specifically to deal with the challenges associated with multidimensional arrays.

Much of the climate/earth/ocean/atmosphere data that you can access will be in the form of netCDF files. They typically have the extension `.nc`, so like `ocean_temps.nc`.

NetCDF files are machine independent, meaning that macs, PCs, linux machines, you name it, they can all read the files.

Also, the netCDF files are self contained - i.e. they carry all the information about the data they contain with them. So they are 'self-describing' like the datasets and dataArrays we have been building.

In fact, Xarray is basically a package devoted to reading, writing, and manipulating netCDFs. This means it's a super easy and useful way to work with geophysical data from nearly anywhere.

In this lesson we are going to use Xarray to load some Sea Surface Temperature data from a netCDF file. We will see how easy it is to make calculations and plots of these big data sets using Xarray.

## credit

This lesson is from Abernathy's book: ([https://earth-env-data-science.github.io/lectures/xarray/xarray\\_intro.html](https://earth-env-data-science.github.io/lectures/xarray/xarray_intro.html)).

# Loading netCDF datasets

The primary tool in the Xarray library that we will use with netCDF files is `xr.open_dataset()`. This will read in a netCDF file and create one of our DataArrays.

In this example we are going to read in a Sea Surface Temperature dataset created by NOAA that goes back to the 1800's. You can learn more about the data here: <https://www.ncdc.noaa.gov/data-access/marineocean-data/extended-reconstructed-sea-surface-temperature-ersst-v5>

First, let's do our normal import statements that we need to access the libraries in this new notebook:

```
In [32]: # do our imports
import xarray as xr
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Let's load in the data using `xr.open_dataset()` and take a look at it:

```
In [33]: # load in the data with open_dataset()







url = 'http://www.esrl.noaa.gov/psd/thredds/dodsC/Datasets/noaa.ersst.v5/sst
ds = xr.open_dataset(url, drop_variables=['time_bnds'])
# ds = xr.open_dataset('../data/NOAA_ERSSTv5_monthly.nc')
```

```
ds
```



```
Out[33]: xarray.Dataset
```

► Dimensions: (lat: 89, lon: 180, time: 2035)

▼ Coordinates:

|             |        |                |                              |   |   |
|-------------|--------|----------------|------------------------------|---|---|
| <b>lat</b>  | (lat)  | float32        | 88.0 86.0 84.0 ... -86.0 ... |  |  |
| <b>lon</b>  | (lon)  | float32        | 0.0 2.0 4.0 ... 354.0 356... |  |  |
| <b>time</b> | (time) | datetime64[ns] | 1854-01-01 ... 2023-07...    |  |  |

▼ Data variables:

|            |                  |         |     |   |   |
|------------|------------------|---------|-----|---|---|
| <b>sst</b> | (time, lat, lon) | float32 | ... |  |  |
|------------|------------------|---------|-----|---|---|

► Indexes: (3)

► Attributes: (39)

Did that work? There is a lot of information there. Let's go through all of it to make sure we understand what our Dataset looks like.

Draw on the board and answer the following:

- What are the dimensions of the data?
- What is the data itself
- what do the coordinate of the dimensions look like?
- draw a schematic of the data and label all the 'sides'
- what is the stuff in the attributes?

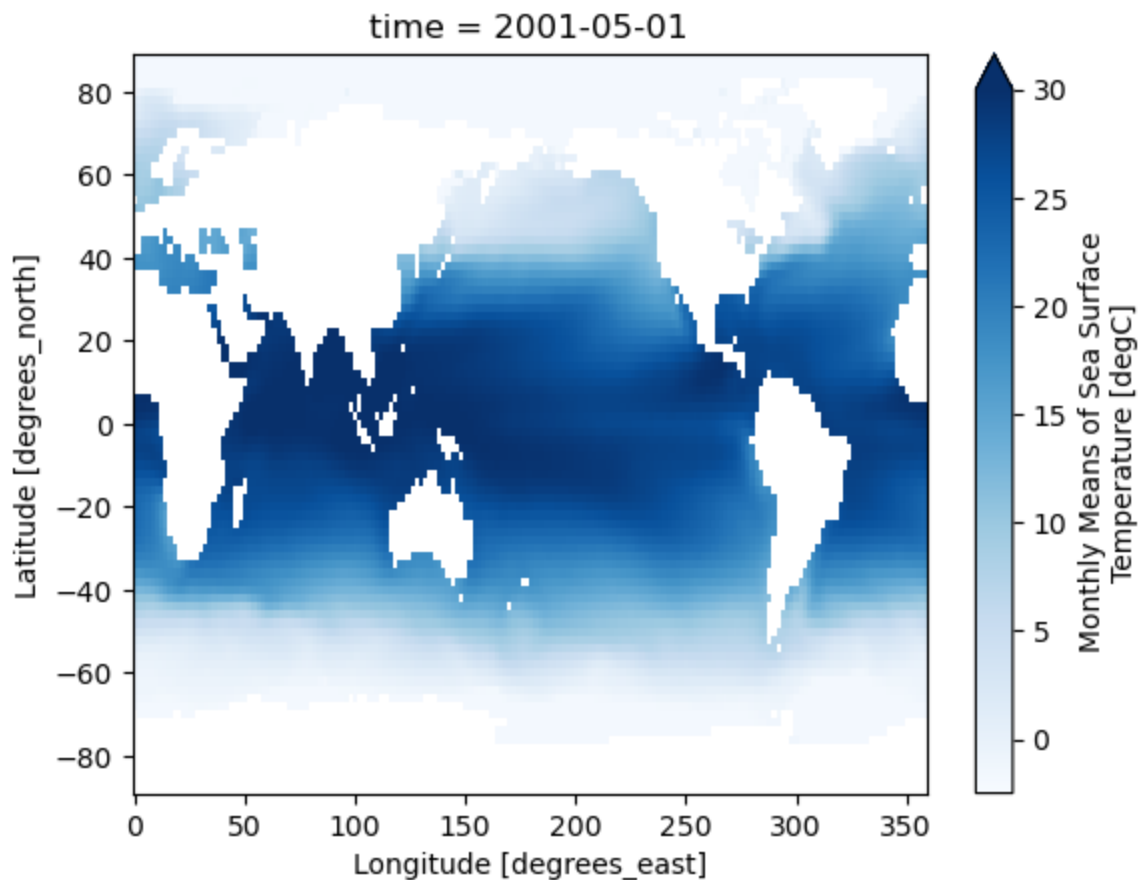
## plotting netcdf data

Next let's make some plots to look at the data. We have lat, lon, Sea Surface Temperature data over a range of times. Maybe let's start with a simple plot of the SST all over the globe on one particular day. What is a good day?

*note* if you look at the time dimension, we see that the data is reported in monthly means with dates on the first of the month - let's pick the first day of a month.

```
In [34]: ds.sst.sel(time = '2001-05-01').plot( vmin=-2.5, vmax=30, cmap='Blues')  
# ds.sst.sel(time = '2001-05-01').plot()
```

```
Out[34]: <matplotlib.collections.QuadMesh at 0x17d02a010>
```



what if we pick a different day of the month?

```
In [35]: # ds.sst.sel(time = '2001-05-15').plot( vmin=-2.5, vmax=30)
```

We got an error because we asked for a specific day that isn't in the dataset. We can get around this sort of thing luckily!

## Nearest point indexing, or 'nearest neighbor lookups'







In the case above we input an exact date that is available in our data. What if we didn't know all the exact dates? Try putting a random date in to the plot call. What if we want to get the time closest to some date we care about? Xarray can handle this if we give it an extra argument using `method='nearest'` :

```
In [36]: ds.sst.sel(time='12-20-1983', method='nearest').plot(vmin=-2.5, vmax=30)
ds
```



Out[36]: xarray.Dataset

► Dimensions: (lat: 89, lon: 180, time: 2035)

▼ Coordinates:

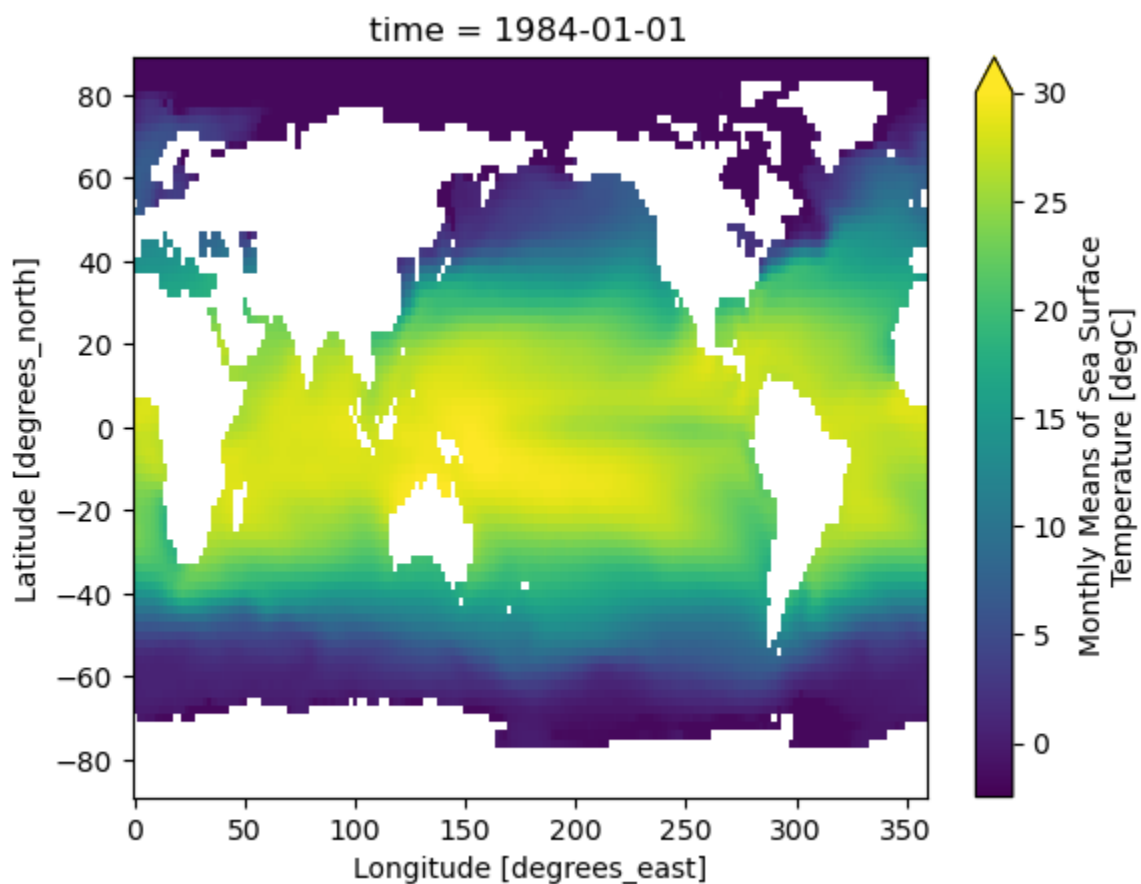
|      |        |                |                              |   |
|------|--------|----------------|------------------------------|---|
| lat  | (lat)  | float32        | 88.0 86.0 84.0 ... -86.0 ... |   |
| lon  | (lon)  | float32        | 0.0 2.0 4.0 ... 354.0 356... |   |
| time | (time) | datetime64[ns] | 1854-01-01 ... 2023-07...    |   |

▼ Data variables:

|     |                  |         |     |   |
|-----|------------------|---------|-----|---|
| sst | (time, lat, lon) | float32 | ... |   |
|-----|------------------|---------|-----|---|

► Indexes: (3)

► Attributes: (39)



Ok, so we can pretty easily make a plot of global SST on a single day. That is pretty cool.

We can use this dataset to see some amazing things without doing a lot of hard work thanks to the people who developed xarray (and the people who created/collected the data!!!!!!).

Let's make a simple plot to see how global average sea surface temperature has changed over time. Do you think we will be

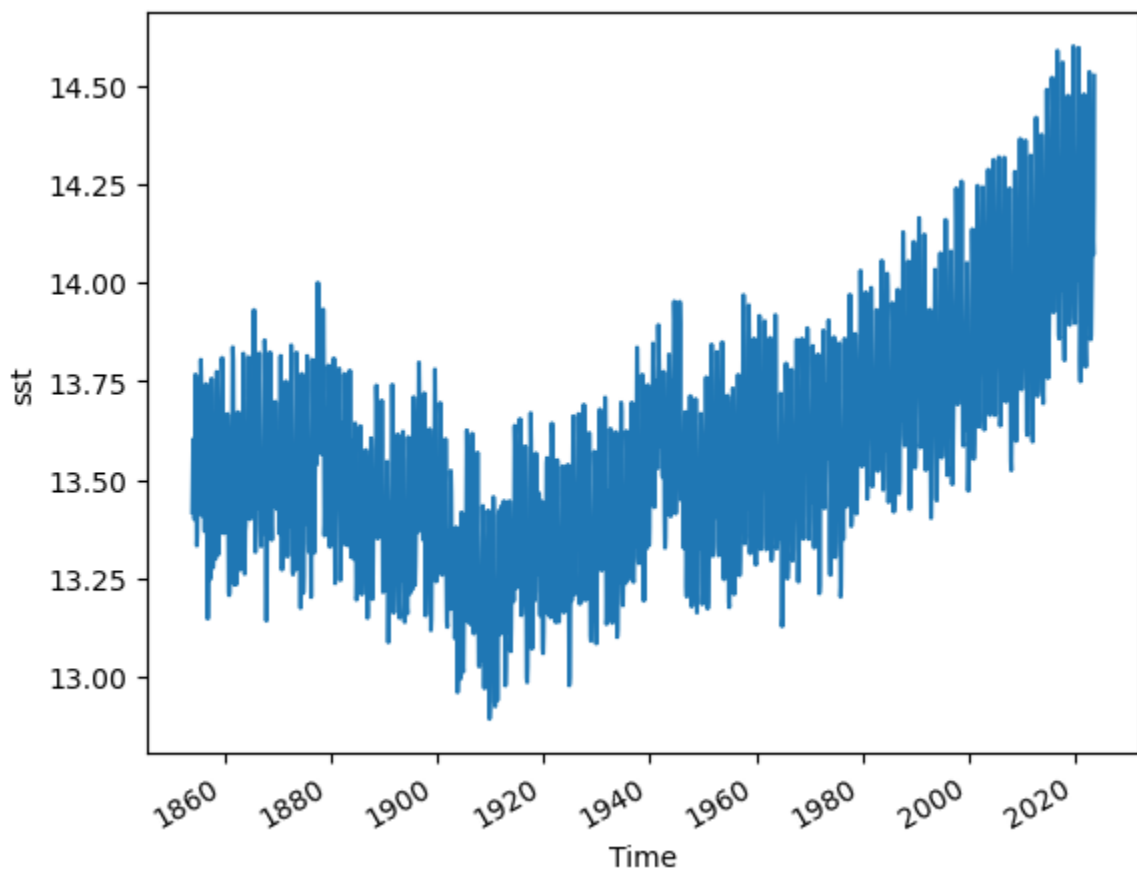
## able to see a warming signal?

To do this we want to use xarray's `.mean()` function. But we need to tell it what kind of mean we want. In other words we need to define the dimensions over which to take the mean. If we are interested in making a plot that shows global averaged sea surface temperature over time, what are the dimensions to average over?

we are going to do something like: `ds.sst.mean(dim=('lat', ...)).plot()` fill in the blanks:

```
In [37]: ds.sst.mean(dim=('lat', 'lon')).plot()  
# ds.ssta.mean(dim=('lat', 'lon')).plot()
```

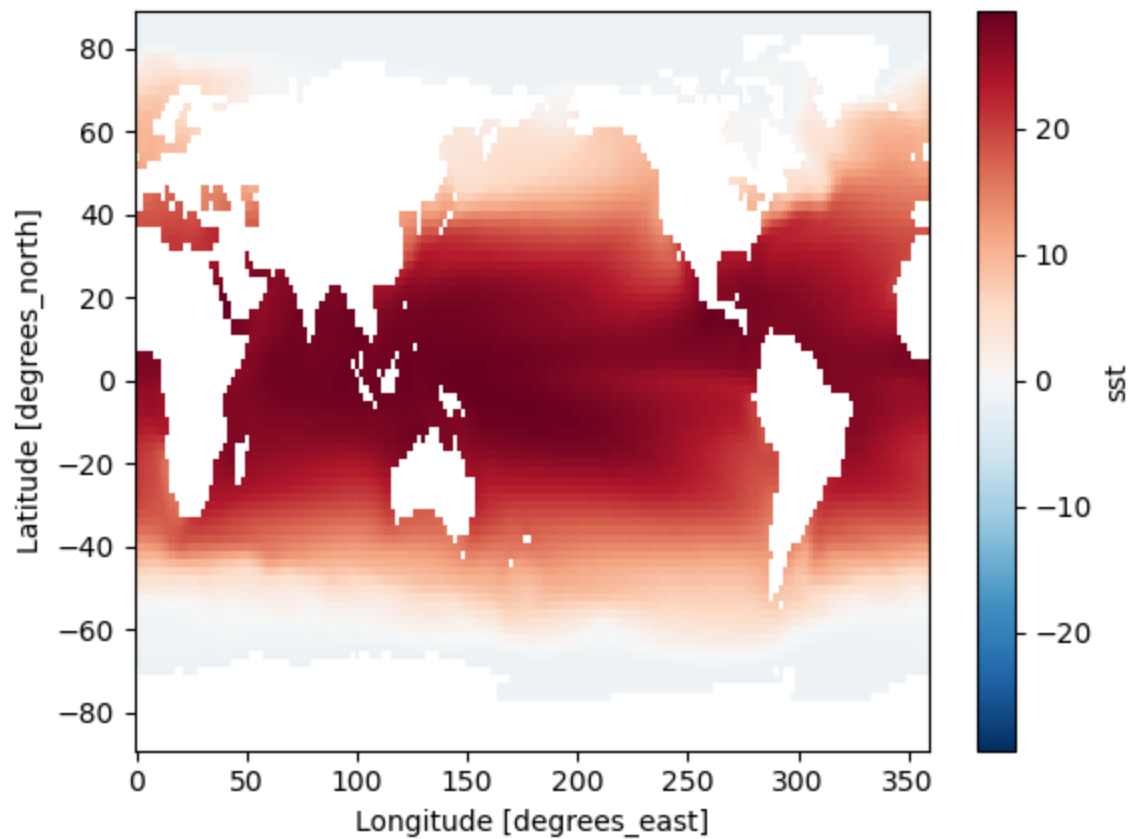
```
Out[37]: [matplotlib.lines.Line2D at 0x17d33c410]
```



What about just plotting the time average map of SST? What dimensions are we going to average over here?

```
In [38]: ds.sst.mean(dim=('time')).plot();
```

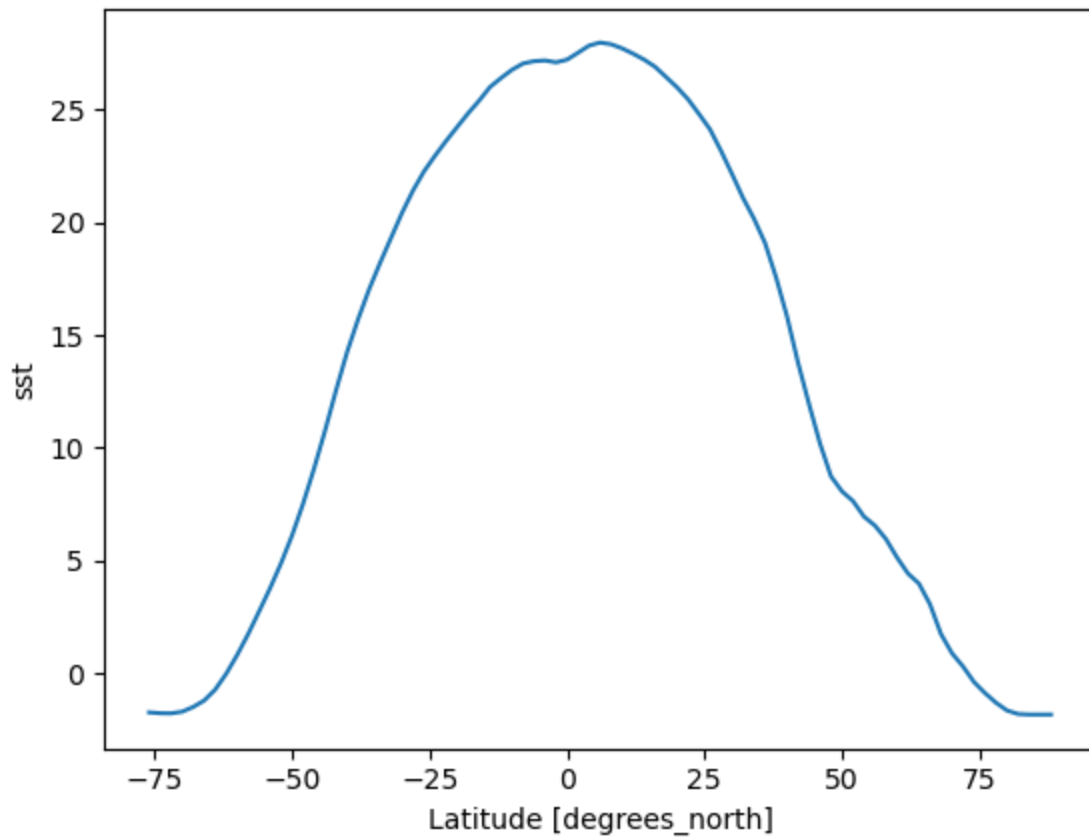




What about the average temperature as a function of latitude? We want to make a line plot that shows how temperature depends on latitude only, how would we do that?

```
In [39]: ds.sst.mean(dim=('lon', 'time')).plot()
```

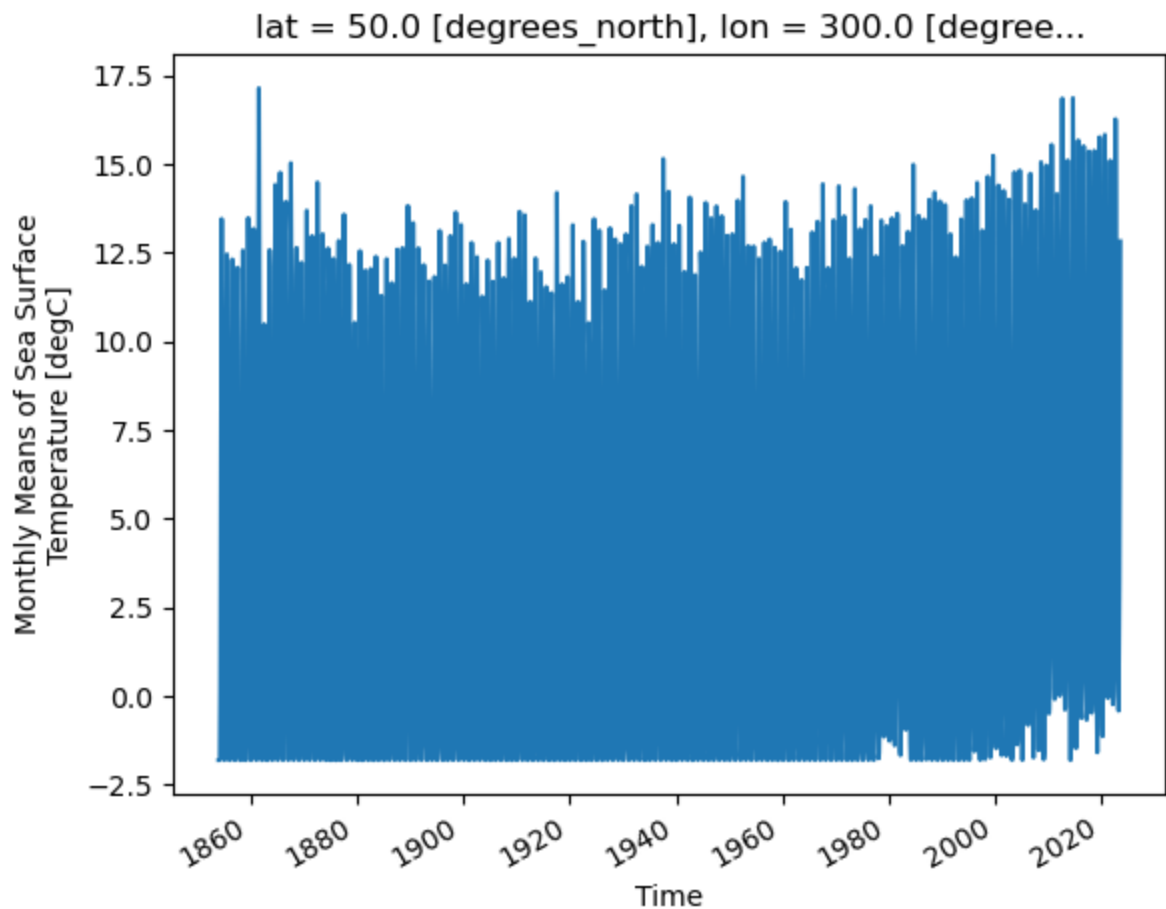
```
Out[39]: [<matplotlib.lines.Line2D at 0x17d454790>]
```



How about a timeseries of temperature at a single point? Let's make a plot of the SST at 45 degrees north, and 230 degrees. How do we do that? Recall the `.sel()` method, and its argument `nearest`

```
In [40]: ds.sst.sel( lon=300, lat=50, method='nearest').plot()
```

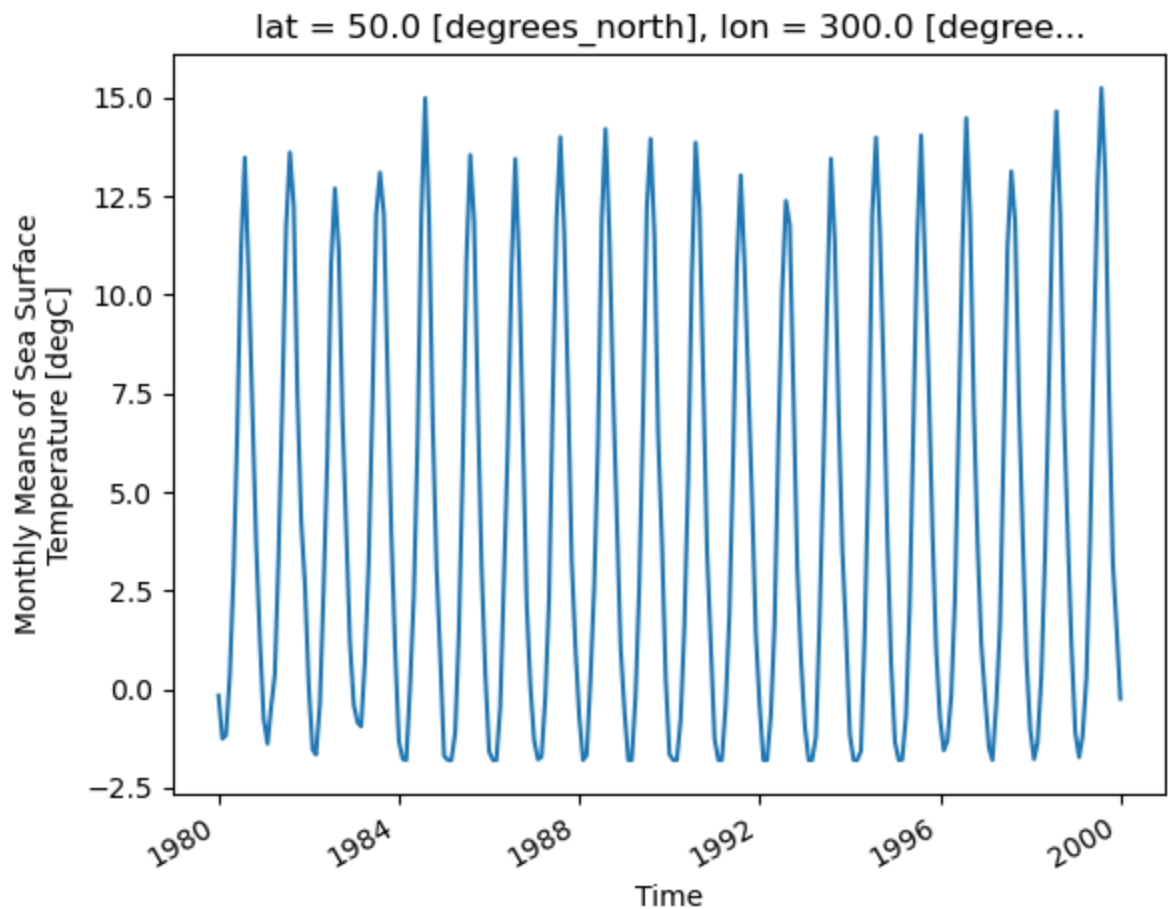
```
Out[40]: [<matplotlib.lines.Line2D at 0x17d511ed0>]
```



that is a mess. Let's adjust the axis so we can see what is happening in that blue mess. Let's pick 20 years of data, from 1980 to 2000 and zoom in. We can do this by setting the range of the x axis. We are going to build up a lot of tricks to make plots look the way we want. This is one.

```
In [41]: # ds.sst.sel( lon=300, lat=50, method='nearest').plot()  
ds.sst.sel(time=slice('1980-01-01','2000-01-01'), lon=300, lat=50).plot()  
# plt.xlim(['1-1-1980', '1-1-2000'])
```

```
Out[41]: [<matplotlib.lines.Line2D at 0x17d580490>]
```

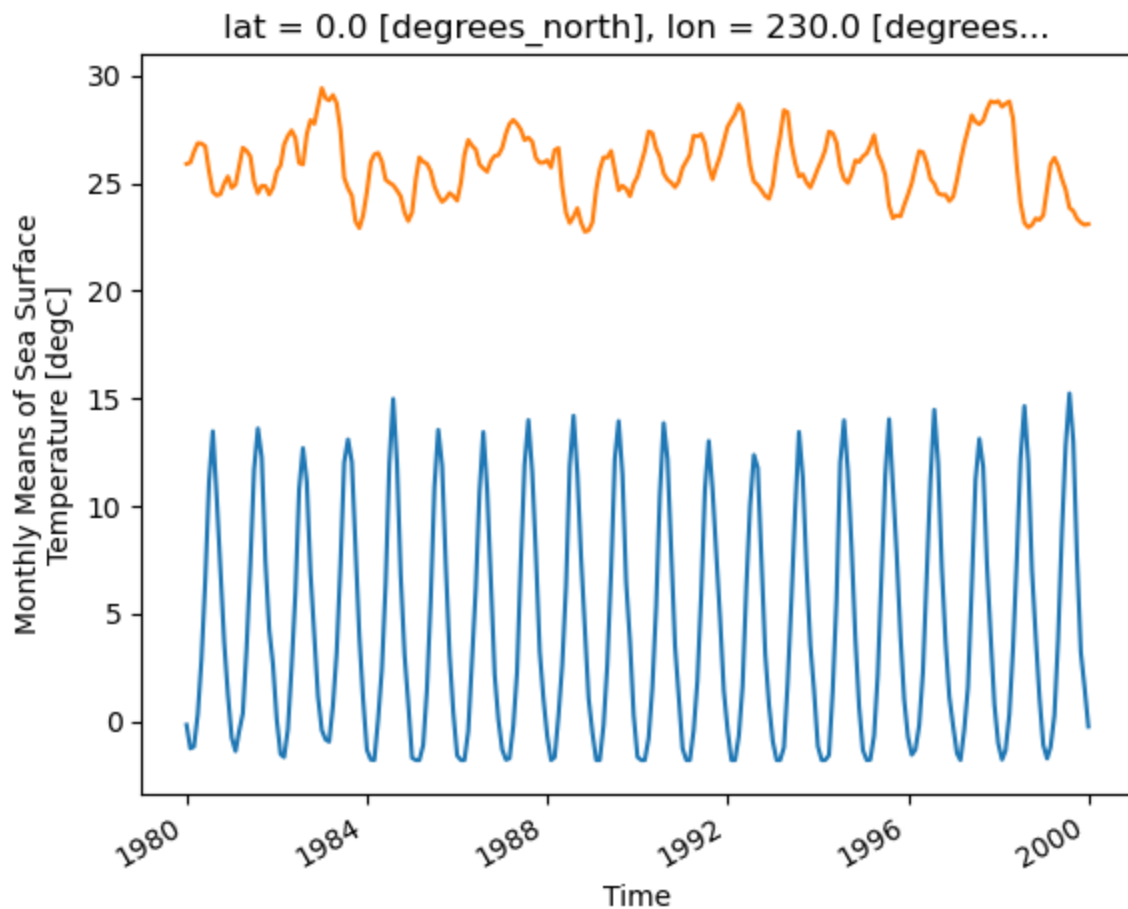


Huh. That's cool. What are we seeing here?

Let's plot two different Latitudes, one high lat and one on the equator:

```
In [42]: ds.sst.sel(time=slice('1980-01-01','2000-01-01'), lon=300, lat=50).plot()  
ds.sst.sel(time=slice('1980-01-01','2000-01-01'), lat=0, lon=230).plot()  
# plt.xlim(['1-1-1980', '1-1-2000'])
```

```
Out[42]: [<matplotlib.lines.Line2D at 0x17d353110>]
```



## Groupby

Yep, we can do groupby here too.

Let's groupby month and apply a mean. This will give us a climatology of SST from the past couple hundred years at every point on the globe:

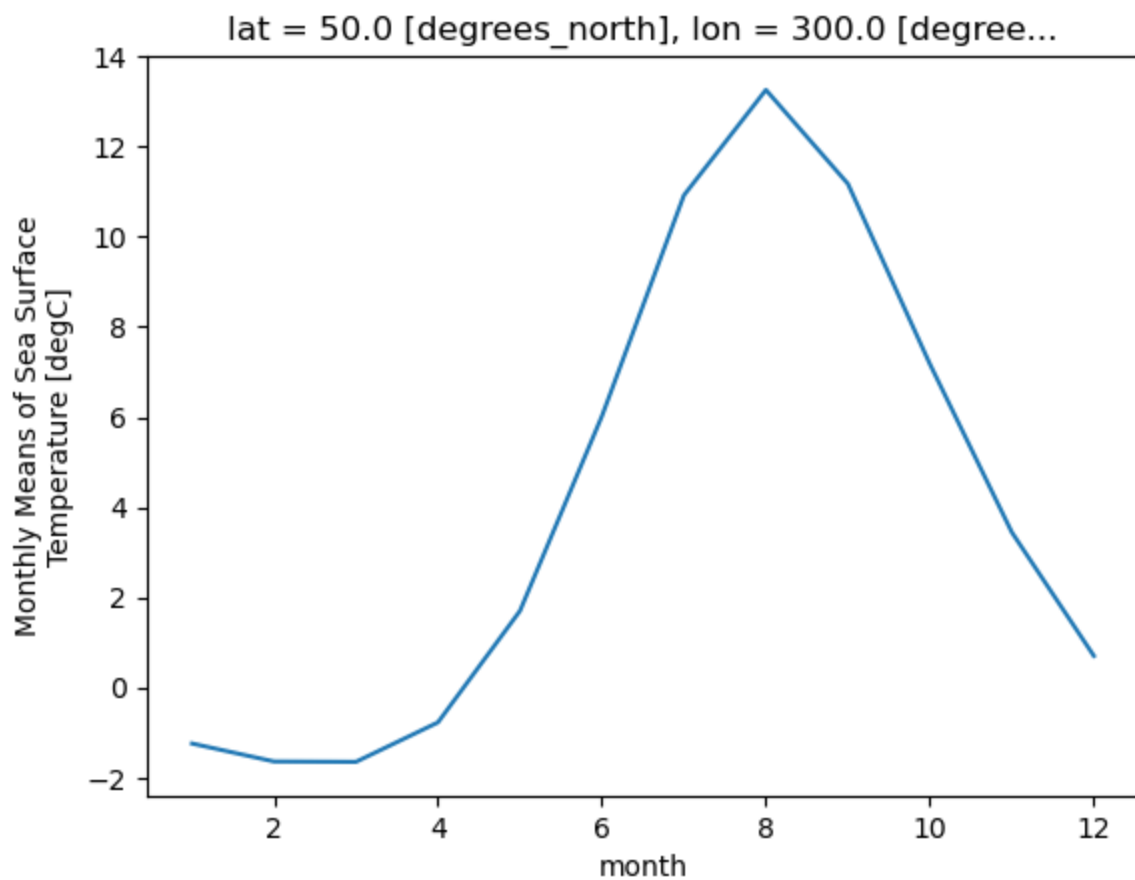
```
In [43]: # group by month
gb = ds.groupby('time.month')

# apply a time mean to the groups to get a monthly mean climatology dataset
ds_mm = gb.mean(dim='time')
```

climatology at a specific point in the North Atlantic:

```
In [44]: ds_mm.sst.sel(lon=300, lat=50).plot()
```

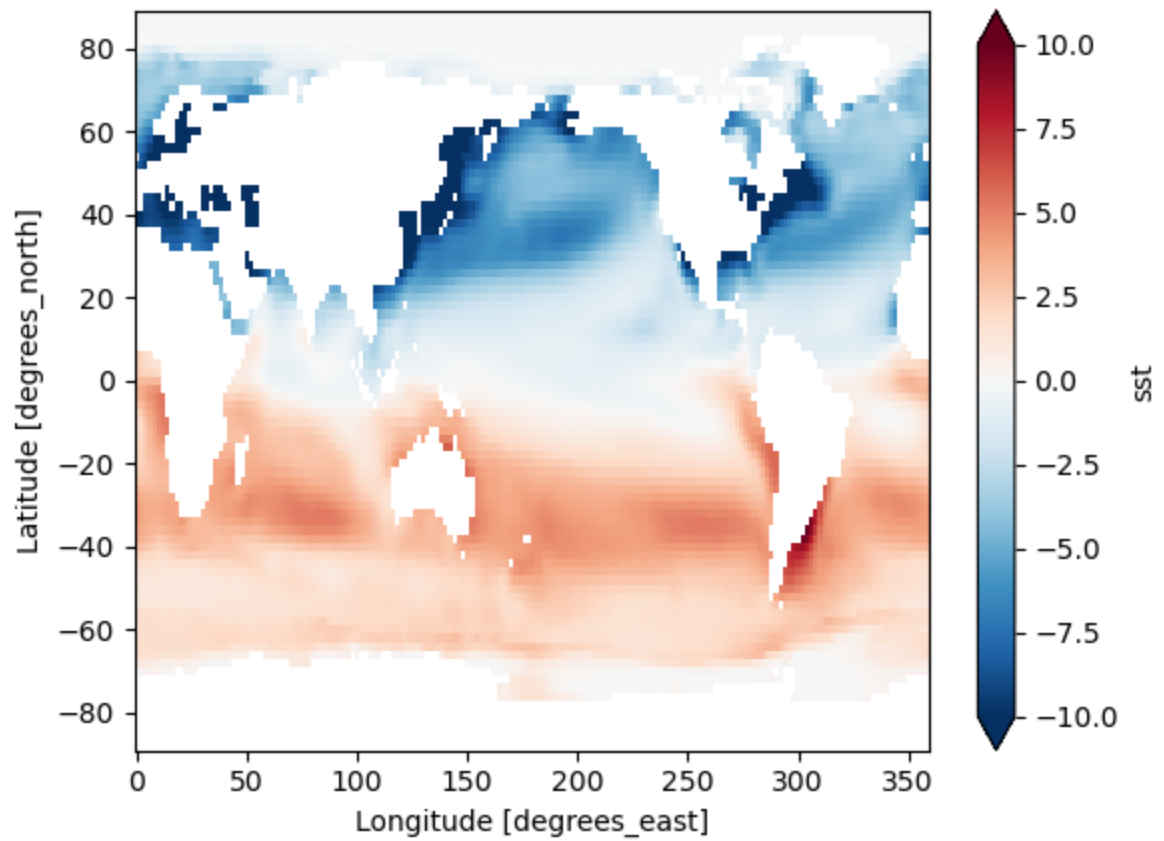
```
Out[44]: [<matplotlib.lines.Line2D at 0x17d2ef4d0>]
```



Plot the July minus Jan differences

```
In [45]: seasonal_diff = ds_mm.sst.sel(month=1) - ds_mm.sst.sel(month=7)
seasonal_diff.plot(vmax=10)
```

```
Out[45]: <matplotlib.collections.QuadMesh at 0x17abe2790>
```



## remove a time mean

Let's look more clearly at the long term SST trend by removing the seasonal climatology

```
In [46]: gb = ds.groupby('time.month')



ds_anom = gb - gb.mean(dim='time')

ds_anom
```



Out[46]: xarray.Dataset

► Dimensions: (lat: 89, lon: 180, time: 2035)

▼ Coordinates:

|             |        |                |                                |   |   |
|-------------|--------|----------------|--------------------------------|---|---|
| <b>lat</b>  | (lat)  | float32        | 88.0 86.0 84.0 ... -86.0 ...   |  |  |
| <b>lon</b>  | (lon)  | float32        | 0.0 2.0 4.0 ... 354.0 356...   |  |  |
| <b>time</b> | (time) | datetime64[ns] | 1854-01-01 ... 2023-07...      |  |  |
| month       | (time) | int64          | 1 2 3 4 5 6 7 8 ... 1 2 3 4... |  |  |

▼ Data variables:

|            |                  |         |                             |   |   |
|------------|------------------|---------|-----------------------------|---|---|
| <b>sst</b> | (time, lat, lon) | float32 | -1.192e-07 -1.192e-07 ..... |  |  |
|------------|------------------|---------|-----------------------------|---|---|

► Indexes: (3)

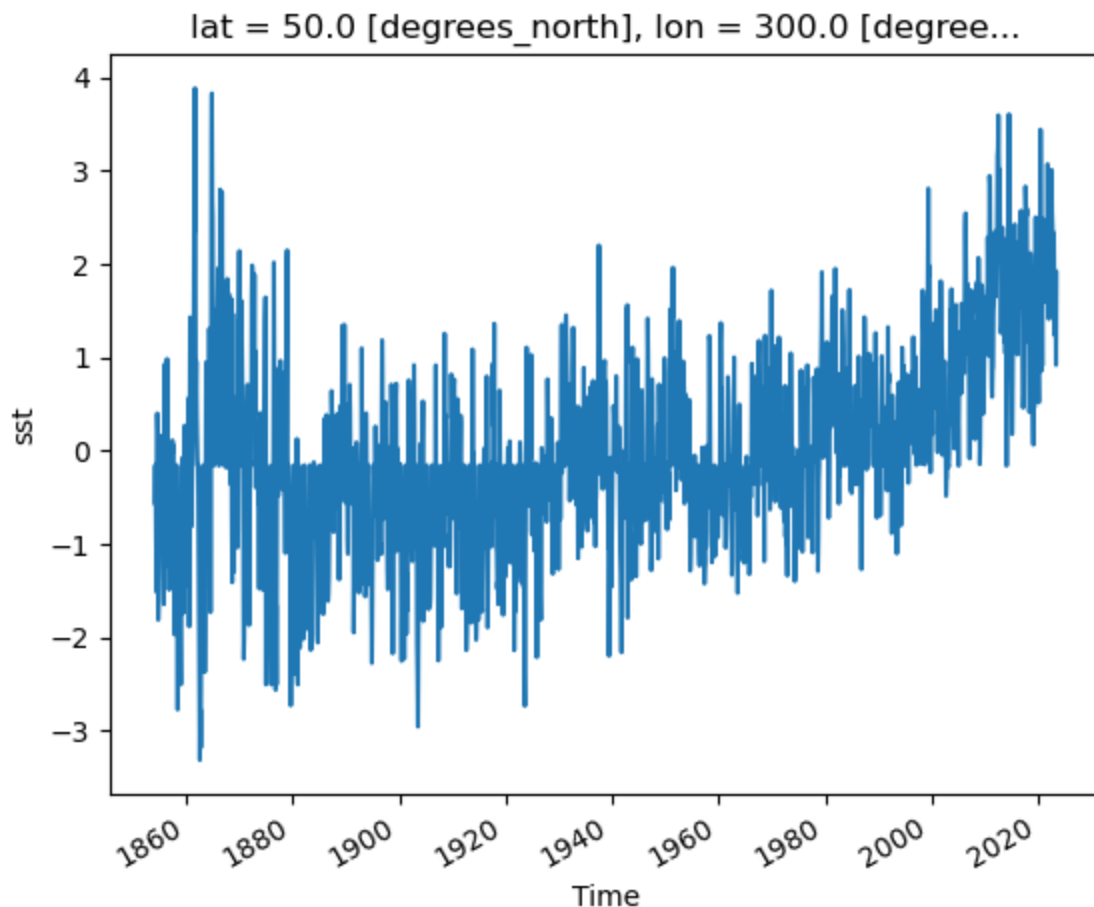
► Attributes: (0)

timeseries of SST anomaly at a certain point:

```
In [47]: ds_anom.sst.sel(lon=300, lat=50).plot()
```

Out[47]: [<matplotlib.lines.Line2D at 0x17a6a0490>]





## Saving data to netcdf

Suppose we are always working with the mean surface temperature. Here calculating the mean is fast, but suppose it were very slow... It would be useful to save the mean data so we don't have to repeat the calculation.

Xarray makes that very easy. In general it works like this:

```
name = "whatever.nc"
some_dataset.to_netcdf(name)
So lets try that for our data:
```

```
In [48]: # generate mean over latitude and longitude
sst_mean = ds.sst.mean(dim=('lat', 'lon'))

name = "sst_mean.nc"

sst_mean.to_netcdf(name)
```

The end...

## Breakout / exercise 03

### Exercise 03 - opening an online dataset

#### NASA ocean color data

This problem demonstrates how we can access Ocean Color data from NASA.

The main repository for NASAs ocean color data is: [https://oceandata.sci.gsfc.nasa.gov/opensdap/](https://oceandata.sci.gsfc.nasa.gov/.opendap/)

We will look at data from the MODIS-Aqua (MODIS-A) satellite, and in particular we will look at the level 3 product, which is data that has gone through the highest level of processing and nicely gridded.

NASA organizes data by year and year day. You can see this structure by clicking through the OpenDAP server. The file used in this example contains the mapped chlorophyll-a data for July 28 (year day 210), 2019.

```
In [49]: import xarray as xr
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

#### Use `xr.open_dataset()` to access the data

The data for yearday 210 of 2019 is located at:

```
url = 'https://oceandata.sci.gsfc.nasa.gov:443/opensdap/MODISA/L3SMI
/2019/210/A2019210.L3m_DAY_CHL_chlor_a_4km.nc'
```





```
In [50]: url = '../04_cartopy/data/A2019210.L3m_DAY_CHL_chlor_a_4km.nc'
data = xr.open_dataset(url)

data
```





Out [50]: xarray.Dataset

► Dimensions: (lat: 4320, lon: 8640, rgb: 3, eightbitcolor: 256)

▼ Coordinates:

|     |       |         |                                |   |   |
|-----|-------|---------|--------------------------------|---|---|
| lat | (lat) | float32 | 89.98 89.94 89.9 ... -89.94... |  |  |
| lon | (lon) | float32 | -180.0 -179.9 ... 179.9 180.0  |  |  |

▼ Data variables:

|         |                      |         |     |   |   |
|---------|----------------------|---------|-----|---|---|
| chlor_a | (lat, lon)           | float32 | ... |  |  |
| palette | (rgb, eightbitcolor) | uint8   | ... |  |  |

► Indexes: (2)

► Attributes: (63)

## Subset the data

let's just grab the mid-atlantic bight.

Note, that for some reason I don't understand, the `lat` `coords` are listed from high to low, so when you slice, you need to reverse the order, i.e. use `sel(lat=slice(41, 38))` *not* `sel(lat=slice(38, 41))`. This is a mystery.

fill in the blanks to get a subset of the data that covers the MAB (the lat boundaries at 38 to 41 degrees, and lon boundaries are -76 to -71):





```
data_mab_nj = data.__( lat = __, lon = __)
```

```
In [51]: data_mab_nj = data.sel( lat=slice(41, 38), lon=slice(-76,-71))
data_mab_nj
```





Out[51]: xarray.Dataset

► Dimensions: (lat: 72, lon: 120, rgb: 3, eightbitcolor: 256)

▼ Coordinates:

|     |       |         |                                  |   |   |
|-----|-------|---------|----------------------------------|---|---|
| lat | (lat) | float32 | 40.98 40.94 40.9 ... 38.06 ...   |  |  |
| lon | (lon) | float32 | -75.98 -75.94 ... -71.06 -71.... |  |  |

▼ Data variables:

|         |                      |         |     |   |   |
|---------|----------------------|---------|-----|---|---|
| chlor_a | (lat, lon)           | float32 | ... |  |  |
| palette | (rgb, eightbitcolor) | uint8   | ... |  |  |

► Indexes: (2)

► Attributes: (63)

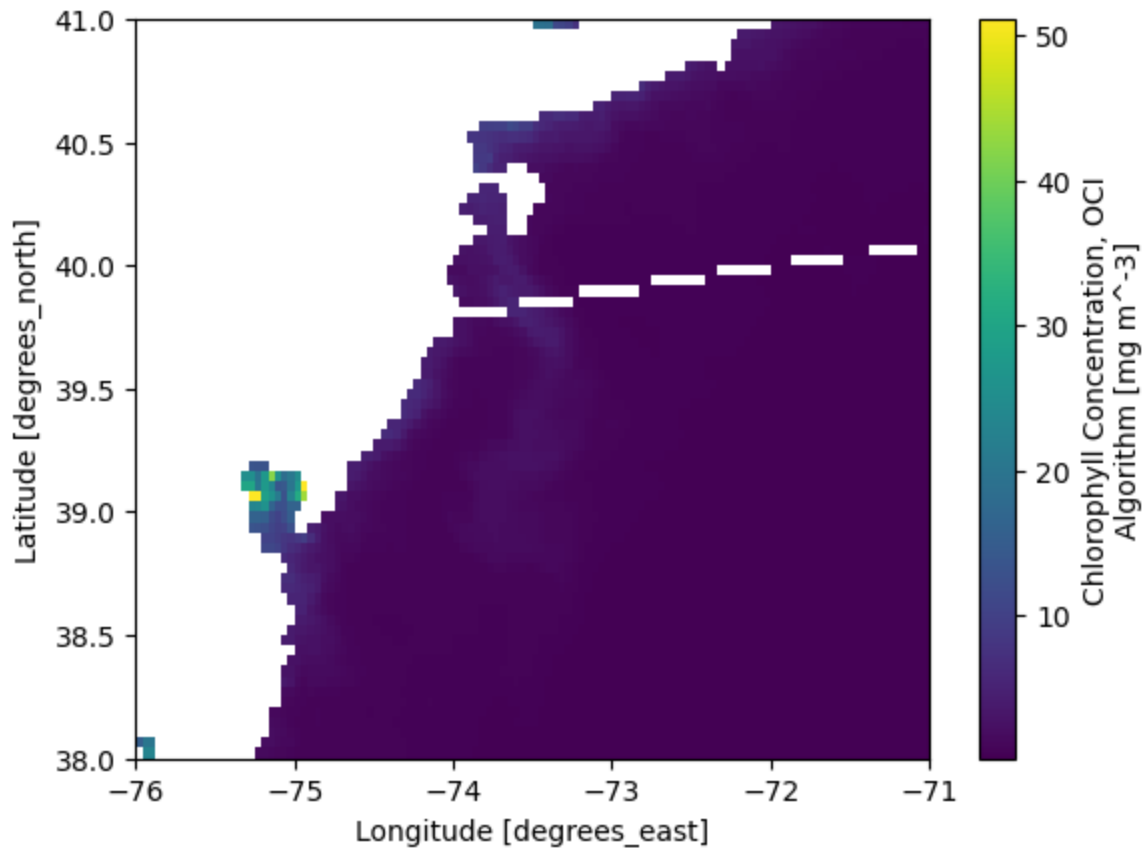
## plot the chlorophyll a

for the mid atlantic bight subset using the built-in xarray plotting routine. i.e. fill in the blanks, and remember we want to just plot the variable `chlor_a`:

```
data_mab_nj.____.____()
```

```
In [52]: data_mab_nj.chlor_a.plot()
```

```
Out[52]: <matplotlib.collections.QuadMesh at 0x17f9db390>
```



## Chla should be plotted on a log scale

Let's make the same plot with matplotlib, and use `np.log10()` to plot the data on a log scale:

```
plt.pcolormesh( data_mab_nj.____, data_mab_nj.____,
np.log10(data_mab_nj.____))
# add a colorbar
```

be sure to label all your axes

```
In [53]: plt.pcolormesh(data_mab_nj.lon, data_mab_nj.lat, np.log10(data_mab_nj.chlor_
cb = plt.colorbar()

plt.xlabel("Longitude")
plt.ylabel("Latitude")

cb.set_label("log 10 Chla")
```

