

Welcome to the WHOI Summer Math Review in Data Analysis!

This course will be taught in **Python** in the format of a Jupyter notebook, which you will be sent following the course! That way, you can try out the exercises interactively as well, as all of the data will be stored in this format for your later use.

Major Learning Objectives

- Load netCDF files as well as a FASTA file for biological sequences
- Learn skills for plotting and analyzing data in 2D and 3D
- Linear and multinomial regression
- Statistical tests

netCDF data type manipulation

The netCDF data we will be working with come from the [UCAR Community Programs Database](https://www.unidata.ucar.edu/software/netcdf/examples/files.html) (<https://www.unidata.ucar.edu/software/netcdf/examples/files.html>), and include sea surface temperature data which were collected for the purposes of use with the IPCC climate assessment and modeling efforts. The data should all be stored in the same location as this Jupyter notebook for the code to work when you try it out on your own later. It should be in the zip file you receive as part of the course, but it also is freely downloadable from the embedded link.

Note: you will also need to have `netCDF4`, a Python package for working with netCDF data, installed. You can install it either by creating a `conda` environment to use with your Jupyter notebook that has this package available, or by downloading the package directly in Anaconda Navigator.

You can opt for the second option by navigating to the "Environments" tab in Anaconda Navigator, searching for `netCDF4` to add to the currently active environment which you used to launch your Jupyter Notebook instance (which should couple together by default), and following the install dialog.

ANACONDA.NAVIGATOR Signed in as akrinos Sign out

Home Environments Learning Community

Search Environments

base (root) 3 EUKuleleupload _ORCA_jl_ eukulele_2 eukulele_test orthofinder_env orthotest r-miniconda r-reticulate singenv test test2 tymefiles

Create Clone Import Remove

Not installed Channels Update index... netcdf X

Name	T	Description	Version
<input type="checkbox"/> libnetcdf	○		4.7.3
<input checked="" type="checkbox"/> netcdf4	○		1.5.3
<input type="checkbox"/> r-netcdf	○		1.9.1

Install Packages X

The following packages will be modified:

Solving package specifications

Cancel Apply

3 packages available matching "netcdf" 1 package selected

Apply Clear

After you have `netCDF4` installed, you should be able to call the following import commands to make the libraries accessible to your current notebook instance.

In addition to `netCDF4`, we'll also be using `pandas`, `scipy`, `datetime`, `numpy`, `matplotlib` and potentially `plotnine`. If you don't have any of these packages, you can use the same import procedure on Anaconda Navigator.

```
In [1]: 1 import pandas as pd
        2 import numpy as np
        3 import datetime
        4 from scipy import stats
        5 import scipy
        6 from netCDF4 import Dataset
        7 import matplotlib.pyplot as plt
```

```
In [2]: 1 from Bio import SeqIO
```

A primer on `netCDF`

A huge problem in environmental data is **dimensionality**. Because we have at least 4 dimensions - horizontal space on two axes, vertical space, and the axis of time - it can be super unwieldy to deal with data even from a single site. To get a sense of how big these numbers can be, let's consider a patch of ocean 1 meter by 1 meter in size. Let's say we're only interested in phytoplankton.

So, we can say for the sake of argument we only care about the surface one meter of water, but we need subdivisions of *micrometers* to say anything useful about the critters we're interested in. We need to observe the evolution of a phytoplankton bloom over 10 days, and we need time points every 1 hour to track diversity. All of the sudden, we have:

$$\text{height} \times \text{width} \times \text{depth} \times \text{depth} = \left(1 \text{ m} \times \frac{10^6 \mu\text{m}}{\text{m}}\right) \times \left(1 \text{ m} \times \frac{10^6 \mu\text{m}}{\text{m}}\right) \times \left(1 \text{ m} \times \frac{10^6 \mu\text{m}}{\text{m}}\right) \times \left(10 \text{ days}\right) \\ 2.40 \times 10^{20} \text{ hourly snapshots of 1 micrometer size}$$

So...in 1 cubic meter of water, if we use high enough resolution, we can get on the order of 10^{20} data points with *hourly* sampling in a single 10-day cruise. That's before we even try any fancy tricks with data processing. This is why we need `netCDF` !

You are free to store your data as a CSV or other spreadsheet, but it just won't be packed as tidily as a `netCDF` can be, which is why so many earth scientists use this format for their 3-dimensional data. A lot of the physical oceanographic resources that you can find will be stored as a `netCDF` file (you will know this by their `.nc` extension).

Getting started with the `netcdf4` package

If you've worked with `netCDF` files before, you've almost definitely used `MATLAB`. In this course, we'll try out an alternative option: the `netcdf4` package in `Python`. This should help those of you that use `Python` for most everything else in your research!

To get started, we want to perform the analogous operation to the `ncdisp` functionality in `MATLAB`. This prints out the variables inside of our `netCDF` file, so that we have a bit of an idea about how to best access them.

`netcdf4` does *not* have this function. However, we can still check out our variable names. To begin, let's use the `Dataset` package that we loaded.

Loading packages in Python

When we load packages in Python, we can use the `from` directive to specifically load a particular function that we plan to use from inside a function a lot. That's what we did with `Dataset` above:

```
from netCDF4 import Dataset
```

This tells `Python` to be selective, and give us the function that we need from this package. This is the most important function in `netCDF` !

```
In [3]: 1 surface_temperatures = Dataset('data/tos_O1_2001-2002.nc')
```

Remember: you need to make sure that you have the NC file in the `data` directory for this to work!

Let's take a look at what we have inside this `netCDF` file!

First, let's check out the `ncattrs`, which are a grouping of all of the *attributes*, or metadata, that characterize our `netCDF` file.

```
In [4]: 1 surface_temperatures.ncattrs()
```

```
Out[4]: ['title',  
         'institution',  
         'source',  
         'contact',  
         'project_id',  
         'table_id',  
         'experiment_id',  
         'realization',  
         'cmor_version',  
         'Conventions',  
         'history',  
         'references',  
         'comment']
```

We can grab an individual `netCDF` attribute by specifying it as a slot or name within this `Python` object.

```
In [5]: 1 surface_temperatures.title
```

```
Out[5]: 'IPSL model output prepared for IPCC Fourth Assessment SRES A2 experiment'
```

Or, we can get a more global view of all of the variables and data inside of our `netCDF` object.

```
In [6]: 1 print(surface_temperatures)
```

```
<class 'netCDF4._netCDF4.Dataset'>
root group (NETCDF3_CLASSIC data model, file format NETCDF3):
  title: IPSL model output prepared for IPCC Fourth Assessment SRES A2 experiment
  institution: IPSL (Institut Pierre Simon Laplace, Paris, France)
  source: IPSL-CM4_v1 (2003) : atmosphere : LMDZ (IPSL-CM4_IPCC, 96x71x19) ; ocean
ORCA2 (ipsl_cm4_v1_8, 2x2L31); sea ice LIM (ipsl_cm4_v
  contact: Sebastien Denvil, sebastien.denvil@ipsl.jussieu.fr
  project_id: IPCC Fourth Assessment
  table_id: Table 01 (13 November 2004)
  experiment_id: SRES A2 experiment
  realization: 1
  cmor_version: 0.96
  Conventions: CF-1.0
  history: YYYY/MM/JJ: data generated; YYYY/MM/JJ+1 data transformed At 16:37:23
on 01/11/2005, CMOR rewrote data to comply with CF standards and IPCC Fourth Assessm
ent requirements
  references: Dufresne et al, Journal of Climate, 2015, vol XX, p 136
  comment: Test drive
  dimensions(sizes): lon(180), lat(170), time(24), bnds(2)
  variables(dimensions): float64 lon(lon), float64 lon_bnds(lon,bnds), float64 lat
(lat), float64 lat_bnds(lat,bnds), float64 time(time), float64 time_bnds(time,bnds),
float32 tos(time,lat,lon)
  groups:
```

Zooming in on a few of these features

```
<class 'netCDF4._netCDF4.Dataset'>
root group (NETCDF3_CLASSIC data model, file format NETCDF3):
  title: IPSL model output prepared for IPCC Fourth Assessment SRES A2 expe
riment
  institution: IPSL (Institut Pierre Simon Laplace, Paris, France)
  source: IPSL-CM4_v1 (2003) : atmosphere : LMDZ (IPSL-CM4_IPCC, 96x71x19) ;
ocean ORCA2 (ipsl_cm4_v1_8, 2x2L31); sea ice LIM (ipsl_cm4_v
  contact: Sebastien Denvil, sebastien.denvil@ipsl.jussieu.fr
  project_id: IPCC Fourth Assessment
  table_id: Table 01 (13 November 2004)
  experiment_id: SRES A2 experiment
  realization: 1
  cmor_version: 0.96
  Conventions: CF-1.0
  history: YYYY/MM/JJ: data generated; YYYY/MM/JJ+1 data transformed At 16:
37:23 on 01/11/2005, CMOR rewrote data to comply with CF standards and IPCC Fo
urth Assessment requirements
  references: Dufresne et al, Journal of Climate, 2015, vol XX, p 136
  comment: Test drive
  dimensions(sizes): lon(180), lat(170), time(24), bnds(2)
  variables(dimensions): float64 lon(lon), float64 lon_bnds(lon,bnds), float
64 lat(lat), float64 lat_bnds(lat,bnds), float64 time(time), float64 time_bnds
(time,bnds), float32 tos(time,lat,lon)
  groups:
```

A few things jump out right away:

- dimensions(sizes): lon(180), lat(170), time(24), bnds(2)
- variables(dimensions): float64 lon(lon), float64 lon_bnds(lon,bnds), float64 lat(lat), float64 lat_bnds(lat,bnds), float64 time(time), float64 time_bnds(time,bnds), float32 tos(time,lat,lon)

We also have some metadata about where the data came from and what it was used for.

Let's try grabbing out some of these data variables!

```
In [7]: 1 surface_temperatures.variables
```

```
Out[7]: OrderedDict([('lon', <class 'netCDF4._netCDF4.Variable'>
    float64 lon(lon)
        standard_name: longitude
        long_name: longitude
        units: degrees_east
        axis: X
        bounds: lon_bnds
        original_units: degrees_east
    unlimited dimensions:
    current shape = (180,)
    filling on, default _FillValue of 9.969209968386869e+36 used),
('lon_bnds', <class 'netCDF4._netCDF4.Variable'>
    float64 lon_bnds(lon, bnds)
    unlimited dimensions:
    current shape = (180, 2)
    filling on, default _FillValue of 9.969209968386869e+36 used),
('lat', <class 'netCDF4._netCDF4.Variable'>
    float64 lat(lat)
        standard_name: latitude
        long_name: latitude
        units: degrees_north
        axis: Y
        bounds: lat_bnds
        original_units: degrees_north
    unlimited dimensions:
    current shape = (170,)
    filling on, default _FillValue of 9.969209968386869e+36 used),
('lat_bnds', <class 'netCDF4._netCDF4.Variable'>
    float64 lat_bnds(lat, bnds)
    unlimited dimensions:
    current shape = (170, 2)
    filling on, default _FillValue of 9.969209968386869e+36 used),
('time', <class 'netCDF4._netCDF4.Variable'>
    float64 time(time)
        standard_name: time
        long_name: time
        units: days since 2001-1-1
        axis: T
        calendar: 360_day
        bounds: time_bnds
        original_units: seconds since 2001-1-1
    unlimited dimensions: time
    current shape = (24,)
    filling on, default _FillValue of 9.969209968386869e+36 used),
('time_bnds', <class 'netCDF4._netCDF4.Variable'>
    float64 time_bnds(time, bnds)
    unlimited dimensions: time
    current shape = (24, 2)
    filling on, default _FillValue of 9.969209968386869e+36 used),
('tos', <class 'netCDF4._netCDF4.Variable'>
    float32 tos(time, lat, lon)
        standard_name: sea_surface_temperature
        long_name: Sea Surface Temperature
        units: K
        cell_methods: time: mean (interval: 30 minutes)
        _FillValue: 1e+20
        missing_value: 1e+20
        original_name: sosstsst
        original_units: degC
        history: At 16:37:23 on 01/11/2005: CMOR altered the data in t
he following ways: added 2.73150E+02 to yield output units; Cyclical dimension was
```

```

output starting at a different lon;
        unlimited dimensions: time
        current shape = (24, 170, 180)
        filling on))

```

Having a list of variables, we can extract the actual data associated with each variable using *array syntax* in Python .

```
In [8]: 1 surface_temperatures['lat'][:]
```

```

Out[8]: masked_array(data=[-79.5, -78.5, -77.5, -76.5, -75.5, -74.5, -73.5, -72.5,
        -71.5, -70.5, -69.5, -68.5, -67.5, -66.5, -65.5, -64.5,
        -63.5, -62.5, -61.5, -60.5, -59.5, -58.5, -57.5, -56.5,
        -55.5, -54.5, -53.5, -52.5, -51.5, -50.5, -49.5, -48.5,
        -47.5, -46.5, -45.5, -44.5, -43.5, -42.5, -41.5, -40.5,
        -39.5, -38.5, -37.5, -36.5, -35.5, -34.5, -33.5, -32.5,
        -31.5, -30.5, -29.5, -28.5, -27.5, -26.5, -25.5, -24.5,
        -23.5, -22.5, -21.5, -20.5, -19.5, -18.5, -17.5, -16.5,
        -15.5, -14.5, -13.5, -12.5, -11.5, -10.5, -9.5, -8.5,
        -7.5, -6.5, -5.5, -4.5, -3.5, -2.5, -1.5, -0.5,
         0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,
         8.5,  9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5,
        16.5, 17.5, 18.5, 19.5, 20.5, 21.5, 22.5, 23.5,
        24.5, 25.5, 26.5, 27.5, 28.5, 29.5, 30.5, 31.5,
        32.5, 33.5, 34.5, 35.5, 36.5, 37.5, 38.5, 39.5,
        40.5, 41.5, 42.5, 43.5, 44.5, 45.5, 46.5, 47.5,
        48.5, 49.5, 50.5, 51.5, 52.5, 53.5, 54.5, 55.5,
        56.5, 57.5, 58.5, 59.5, 60.5, 61.5, 62.5, 63.5,
        64.5, 65.5, 66.5, 67.5, 68.5, 69.5, 70.5, 71.5,
        72.5, 73.5, 74.5, 75.5, 76.5, 77.5, 78.5, 79.5,
        80.5, 81.5, 82.5, 83.5, 84.5, 85.5, 86.5, 87.5,
        88.5, 89.5],
        mask=False,
        fill_value=1e+20)

```

But **remember**, these are multi-dimensional values (that are compatible with the other values in this `netCDF` file). So we can't just use them like a column in a `DataFrame` , like you might have been introduced to in the `Python` course.

To get started, let's subset our data out (the first time point), so that it's a bit easier to work with.

If you remember from above, our "tos" variable has three dimensions: time, latitude, and longitude, in that order. So we'll take the zero-indexed first entry in `tos` , and then start with all of the latitude and longitude values.

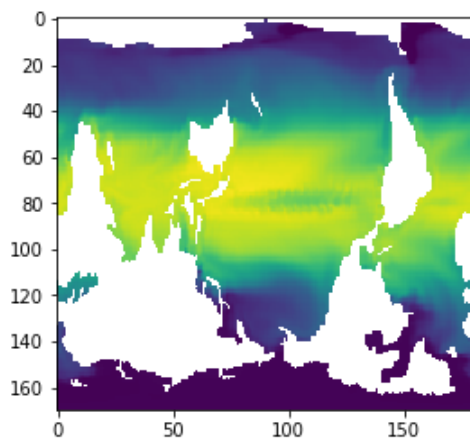
```
In [9]: 1 sst = surface_temperatures['tos'][0,:,:]
```

```
In [10]: 1 sst
```

```
Out[10]: masked_array(
  data=[[--, --, --, ..., --, --, --],
        [--, --, --, ..., --, --, --],
        [--, --, --, ..., --, --, --],
        ...,
        [271.4374694824219, 271.4379577636719, 271.43841552734375, ...,
          271.4361877441406, 271.4365539550781, 271.4369812011719],
        [271.4451904296875, 271.44525146484375, 271.4453125, ...,
          271.445068359375, 271.4450988769531, 271.44512939453125],
        [271.4591979980469, 271.45916748046875, 271.4591369628906, ...,
          271.459228515625, 271.459228515625, 271.4591979980469]],
  mask=[ [ True,  True,  True, ...,  True,  True,  True],
        [ True,  True,  True, ...,  True,  True,  True],
        [ True,  True,  True, ...,  True,  True,  True],
        ...,
        [False, False, False, ..., False, False, False],
        [False, False, False, ..., False, False, False],
        [False, False, False, ..., False, False, False]],
  fill_value=1e+20,
  dtype=float32)
```

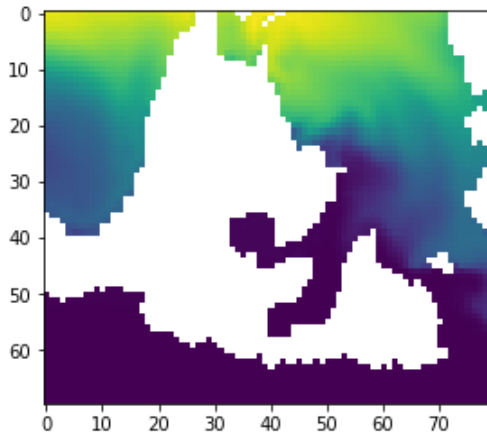
```
In [11]: 1 plt.imshow(sst, interpolation='none')
```

```
Out[11]: <matplotlib.image.AxesImage at 0x1a1f4534a8>
```



This looks recognizable, but it's pretty...upside down. Let's try naïvely plotting out these data for sea surface temperatures (the `tos` variable)! This time, let's try a smaller slice on latitude and longitude.

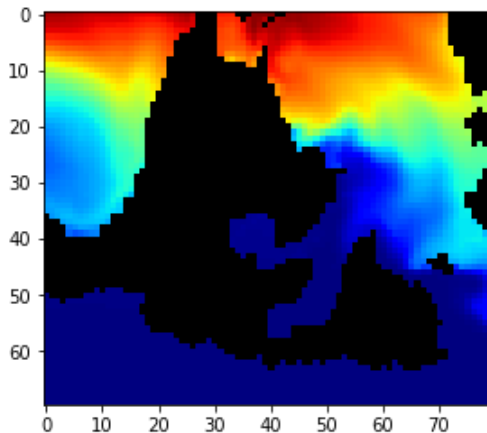

```
In [12]: 1 sst = surface_temperatures['tos'][0,100:400,100:400]
2 plt.imshow(sst, interpolation='none')
3 plt.show()
```



Does anyone recognize this map? Notice that here, we have some null values, which are being masked as white points on the plot above. Let's convert these to black, and use the old friend of the physical oceanographer: the "jetcolors" colormap (though it's important to note that there are a lot of [opinions](https://gorelik.net/2020/08/17/what-is-the-biggest-problem-of-the-jet-and-rainbow-color-maps-and-why-is-it-not-as-evil-as-i-thought/) (<https://gorelik.net/2020/08/17/what-is-the-biggest-problem-of-the-jet-and-rainbow-color-maps-and-why-is-it-not-as-evil-as-i-thought/>) on whether this is a totally terrible colormap to use).

```
In [13]: 1 cmap = plt.cm.jet
2 cmap.set_bad('black',1.) # set the missing points to black
3 plt.imshow(sst, interpolation='nearest', cmap=cmap)
```

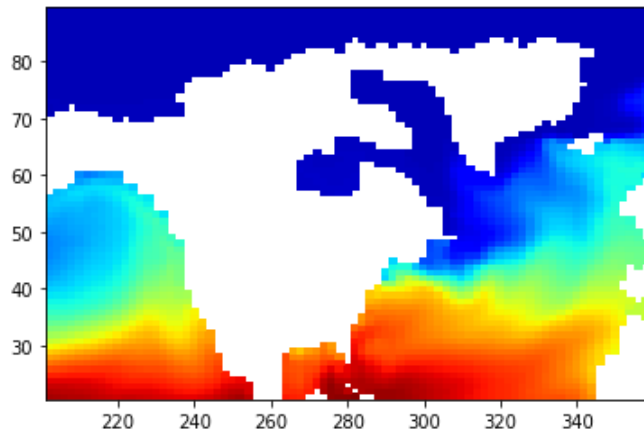
Out[13]: <matplotlib.image.AxesImage at 0x1a1ff39a90>



Is it any clearer now where we might be? To figure it out, let's use the latitude and longitude variables for the same slice of data. For this, we have to combine three different arrays, so we'll use the function `pcolor` to specify different axes.

```
In [14]: 1 lon = surface_temperatures['lon'][100:]
          2 lat = surface_temperatures['lat'][100:]
          3 sst = surface_temperatures['tos'][0, 100:, 100:]
          4
          5 # We use vmin and vmax to set the bounds of the colorbar. Notice that these measur
          6 plt.pcolor(lon, lat, sst, cmap=cmap, vmin=270, vmax=300)
```

Out[14]: <matplotlib.collections.PolyCollection at 0x10a5da470>



Much better! Now we have the first time point of sea surface temperatures for the North Atlantic and Pacific Oceans nearest to the U.S.

```
In [15]: 1 print(surface_temperatures.variables["time"])
          2 surface_temperatures["time"][0] # in days since January 1, 2001.
```

```
<class 'netCDF4._netCDF4.Variable'>
float64 time(time)
  standard_name: time
  long_name: time
  units: days since 2001-1-1
  axis: T
  calendar: 360_day
  bounds: time_bnds
  original_units: seconds since 2001-1-1
unlimited dimensions: time
current shape = (24,)
filling on, default _FillValue of 9.969209968386869e+36 used
```

Out[15]: masked_array(data=15.,
 mask=False,
 fill_value=1e+20)

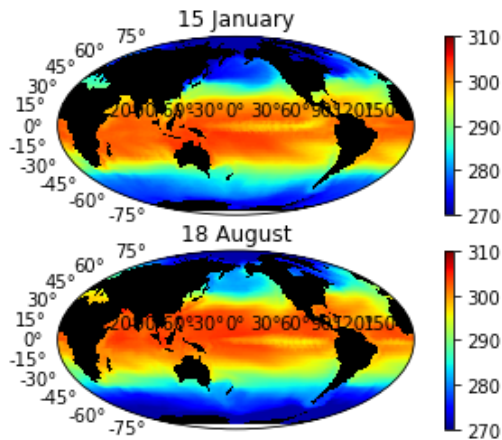
So, the data point we initially pulled is associated with 15 January, 2001. We can get into summer 2001 (August 18th) by taking the fourth data point in this sea surface temperature data and plotting it:

```

In [16]: 1 surface_temperatures["time"][4]
          2
          3 lon = np.subtract(surface_temperatures['lon'],180)
          4 lat = surface_temperatures['lat']
          5 sst = surface_temperatures['tos'][1,:,:]
          6
          7 fig, axs = plt.subplots(ncols=1, nrows=2, subplot_kw={'projection': "mollweide"})
          8
          9 pcm1 = axs[0].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27
10  axs[0].title.set_text('15 January')
11  fig.colorbar(pcm1, ax=axs[0])
12
13  lon = np.subtract(surface_temperatures['lon'],180)
14  lat = surface_temperatures['lat']
15  sst = surface_temperatures['tos'][20,:,:]
16
17  pcm2 = axs[1].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27
18  axs[1].title.set_text('18 August')
19  fig.colorbar(pcm2, ax=axs[1])

```

Out[16]: <matplotlib.colorbar.Colorbar at 0x1a203920f0>



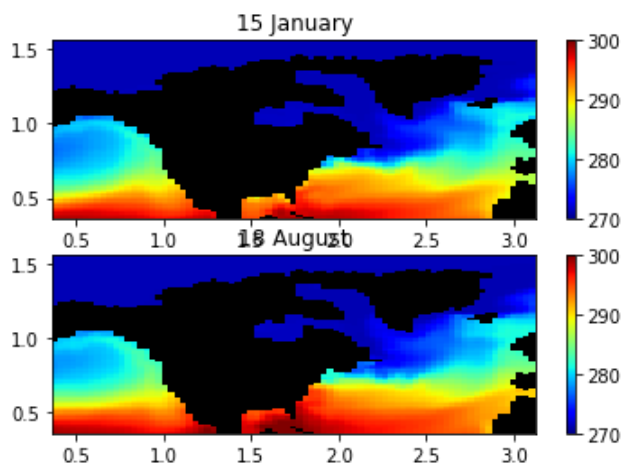
It's a little difficult to see anything here, let's try zooming in a little further.

```

In [17]: 1 lon = surface_temperatures['lon'][100:]
2 lat = surface_temperatures['lat'][100:]
3
4 surface_temperatures["time"][4]
5
6 lon = np.subtract(surface_temperatures['lon'][100:],180)
7 lat = surface_temperatures['lat'][100:]
8 sst = surface_temperatures['tos'][1,100:,100:]
9
10 fig, axs = plt.subplots(ncols=1, nrows=2)
11
12 pcm1 = axs[0].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27)
13 axs[0].title.set_text('15 January')
14 fig.colorbar(pcm1, ax=axs[0])
15
16 lon = np.subtract(surface_temperatures['lon'][100:],180)
17 lat = surface_temperatures['lat'][100:]
18 sst = surface_temperatures['tos'][4,100:,100:]
19
20 pcm2 = axs[1].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27)
21 axs[1].title.set_text('18 August')
22 fig.colorbar(pcm2, ax=axs[1])

```

Out[17]: <matplotlib.colorbar.Colorbar at 0x1a203e9a58>



Note that there are better ways to plot these data, such as `xarray`, but this works for an initial look

We can start to see some differences in the surface temperatures, but overall this is still pretty obscure. Instead, let's go backwards and convert our data to 1-dimensional data. Then, we can calculate the

- mean temperature in each region, for each season
- repeat this process for some delimited regions across every time point in our dataset
- we can convert the numeric time to dates using some handy packages in Python

Selecting and processing regions

We select the following regions of the ocean:

- **REGION 1** - 290-310 longitude, 30-50 latitude (U.S. East Coast)
- **REGION 2** - 230-250 longitude, 30-50 latitude (U.S. West Coast)
- **REGION 3** - 320-350 longitude, 60-70 latitude (Greenland Sea)

To take the average over the full area for each time point in the sampling period.

```

In [18]: 1  ## REGION 1
2  long_indices = np.where((surface_temperatures['lon'][:] > 290) & (surface_temperat
3  lon = np.subtract(surface_temperatures['lon'][long_indices],180)
4  lat_indices = np.where((surface_temperatures['lat'][:] > 30) & (surface_temperatur
5  lat = surface_temperatures['lat'][lat_indices]
6  sst = surface_temperatures['tos'][:,lat_indices[0],long_indices[0]]
7
8  # take mean over axis 1&2 (axis 0 is time)
9  mean_sst_region1 = np.mean(sst,(1, 2))
10
11  ## REGION 2
12  long_indices = np.where((surface_temperatures['lon'][:] > 230) & (surface_temperat
13  lon = np.subtract(surface_temperatures['lon'][long_indices],180)
14  lat_indices = np.where((surface_temperatures['lat'][:] > 30) & (surface_temperatur
15  lat = surface_temperatures['lat'][lat_indices]
16  sst = surface_temperatures['tos'][:,lat_indices[0],long_indices[0]]
17
18  # take mean over axis 1&2 (axis 0 is time)
19  mean_sst_region2 = np.mean(sst,(1, 2))
20
21  ## REGION 3
22  long_indices = np.where((surface_temperatures['lon'][:] > 320) & (surface_temperat
23  lon = np.subtract(surface_temperatures['lon'][long_indices],180)
24  lat_indices = np.where((surface_temperatures['lat'][:] > 60) & (surface_temperatur
25  lat = surface_temperatures['lat'][lat_indices]
26  sst = surface_temperatures['tos'][:,lat_indices[0],long_indices[0]]
27
28  # take mean over axis 1&2 (axis 0 is time)
29  mean_sst_region3 = np.mean(sst,(1, 2))

```

Now, we'll play with converting the time over, using the `datetime` package in Python.

```

In [19]: 1  times_all = surface_temperatures['time'][:].data # all of the times in days

In [20]: 1  date_start = datetime.datetime.strptime("1/1/2001", "%m/%d/%Y") # the start date i
2
3  timepoint_dates = [date_start + datetime.timedelta(days=curr) for curr in times_al

```

Now, let's organize all of our data into a tidy `pandas DataFrame` to work with it more easily!

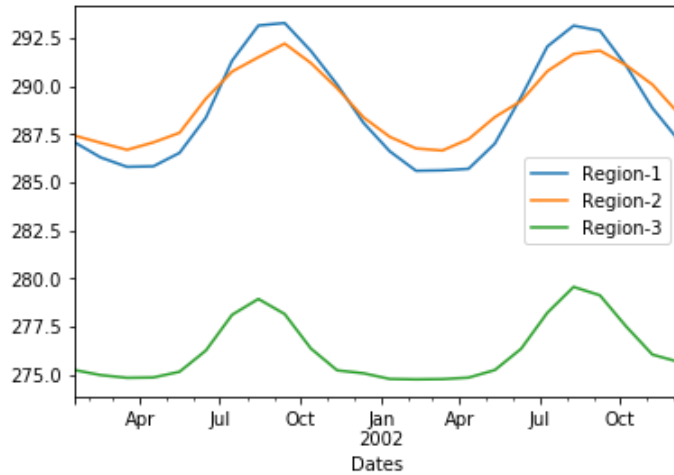
```
In [21]: 1 sst_summary = pd.DataFrame({"Dates": timepoint_dates,
2                                     "Region-1": mean_sst_region1,
3                                     "Region-2": mean_sst_region2,
4                                     "Region-3": mean_sst_region3})
5 sst_summary
```

Out[21]:

	Dates	Region-1	Region-2	Region-3
0	2001-01-16	287.076007	287.421424	275.239787
1	2001-02-15	286.282215	287.044196	274.976033
2	2001-03-17	285.796593	286.681440	274.834613
3	2001-04-16	285.823866	287.064503	274.852754
4	2001-05-16	286.523908	287.566181	275.151251
5	2001-06-15	288.351963	289.338642	276.250579
6	2001-07-15	291.312688	290.748498	278.110964
7	2001-08-14	293.141025	291.492137	278.932137
8	2001-09-13	293.269814	292.199219	278.148835
9	2001-10-13	291.816924	291.188527	276.353714
10	2001-11-12	290.063347	289.874549	275.221713
11	2001-12-12	288.084102	288.368214	275.071868
12	2002-01-11	286.603845	287.357597	274.775738
13	2002-02-10	285.591185	286.753956	274.752665
14	2002-03-12	285.614270	286.642628	274.767164
15	2002-04-11	285.693336	287.234475	274.844677
16	2002-05-11	287.003436	288.379783	275.238347
17	2002-06-10	289.428205	289.218049	276.341515
18	2002-07-10	292.051511	290.767778	278.216598
19	2002-08-09	293.131989	291.665164	279.558461
20	2002-09-08	292.874812	291.832657	279.122219
21	2002-10-08	291.051322	291.078776	277.501920
22	2002-11-07	288.847562	290.070563	276.040850
23	2002-12-07	287.173381	288.565054	275.660785

```
In [22]: 1 sst_summary.plot(x="Dates")
```

```
Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1fbd3b38>
```



```
In [23]: 1 ## We could also convert our data to a tidier format
2 pd.wide_to_long(sst_summary, stubnames="Region", i="Dates", j="Number", sep="-").reset
```

```
Out[23]:
```

	Dates	Number	Region
0	2001-01-16	1	287.076007
1	2001-02-15	1	286.282215
2	2001-03-17	1	285.796593
3	2001-04-16	1	285.823866
4	2001-05-16	1	286.523908
...
67	2002-08-09	3	279.558461
68	2002-09-08	3	279.122219
69	2002-10-08	3	277.501920
70	2002-11-07	3	276.040850
71	2002-12-07	3	275.660785

72 rows × 3 columns

Answering a research question with these data

Now that we've seen a few ways to manipulate and plot our data, perhaps we would like to give it a go to perform *statistical regression* on our data to establish some trend over space or time.

In this case, we don't really have enough time points to establish any kind of time trend, using the dataset we picked. So instead, let's pick out a few different latitudes and see if we can establish a trend via *linear regression*.

Linear Regression

In linear regression, we have the goal of representing our data with a function in the format:

$$y_{est} = \beta_0 + \beta_1 x + \epsilon$$

Where y_{est} is some estimated response variable, β_1 and β_2 are *parameters* of the equation (you may have seen them expressed as b and m , the y-intercept and slope, respectively), and ϵ is some error, because this regression procedure we're following is imperfect. In our case, because we're going to be modeling the changes in sea surface temperature (in Kelvin) along some latitude axis, we can name SST s and latitude a and have:

$$\hat{s} = \beta_0 + \beta_1 a + \epsilon$$

Using linear algebra to estimate parameters

Where we use the notation \hat{s} to denote that these are *estimated* values for sea surface temperature. Because it wasn't covered in a previous math review, the way that we actually estimate these parameters β_0 and β_1 , the unique values which describe our specific data, is using linear algebra. We can imagine our data in matrix-vector format, where n is the total number of observations that we have:

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{bmatrix} = \begin{bmatrix} 1 & a_0 \\ 1 & a_1 \\ \vdots & \vdots \\ 1 & a_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

Because β_0 is just multiplied by the identity to keep β_0 as the values, and β_1 has as its coefficient the latitude observations that we're using to generate a *straight-line model*.

We can say in general that this equation has the form $\mathbf{s} = \mathbf{E}\mathbf{b}$, where we use a capital letter to denote a matrix, in this case the matrix of the multipliers on our coefficients, and a lowercase letter to represent a vector (our SST predictions and the β coefficients). We use **boldface** for both of these to indicate their multi-dimensionality.

So for every way that we define the parameters, we're going to have some error that we wish to minimize. Because the equation of the prediction is $\beta_0 + \beta_1 a$, this error can be represented as:

$$\text{error}(\beta_0, \beta_1) = \sum_{i=1}^n (s_i - (\beta_0 + \beta_1 a_i))^2$$

So we solve the linear algebra-based matrix-form equation above to make the error expression we just wrote out as *small as possible*. This yields the following formula for the β coefficients:

$$\beta = (E^T E)^{-1} E^T s$$

Which has a derivation that we won't go through here, since we're talking through how to analyze data using available toolkits (but it's a cool process to know!).

Estimating percentage of explained variance using R^2

A metric we often use to express the "percentage of variance explained", which is just a fancy way of how much of the trend we captured using the summary linear model, is called R^2 , or the **coefficient of determination**. The formula for this value is:

$$R^2 = 1 - \frac{\sum_{t=1}^n (s_t - (\beta_0 + \beta_1 a_t))^2}{\sum_{t=1}^n (s_t - \bar{s})^2}$$

So in this case, we're calculating the error associated with our regression process ($\hat{s} = \beta_0 + \beta_1 a_t$) and dividing that by the difference of each one of our dependent variable values (SST measurements) from the *mean* of all of the SST measurements (\bar{s}). This gives us a ratio of how far off we are, relative to how different *all* of the values we have are from the mean. We subtract this ratio from 1 to get how much of the variation *is* explained, as opposed to expressing how far off we are.

Extracting some values to regress on

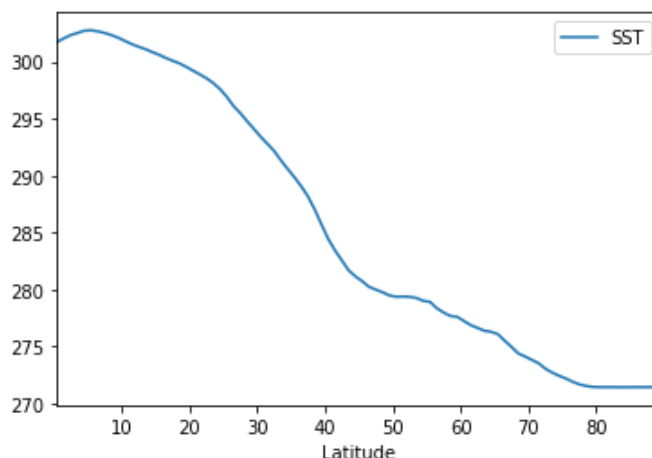
This time, we'll write a *function* in Python in order to extract the mean SST at each latitude over the course of all time points.

```
In [24]: 1 # A function which takes as input a set of latitude and longitude bounds
2 # and then calculates & returns mean SST across time and longitude.
3 def extract_mean_sst_by_lat(min_lat, max_lat, min_lon, max_lon, sst_frame):
4     long_indices = np.where((sst_frame['lon'][:] > min_lon) & \
5                             (sst_frame['lon'][:] < max_lon))
6     lat_indices = np.where((sst_frame['lat'][:] >= min_lat) & \
7                             (sst_frame['lat'][:] <= max_lat)) # use >= so that we c
8
9     lon = np.subtract(sst_frame['lon'][long_indices], 180)
10    lat = sst_frame['lat'][lat_indices]
11    sst = surface_temperatures['tos'][:, lat_indices[0], long_indices[0]]
12
13    mean_sst = np.mean(sst, (0, 2))
14    latitude_frame = pd.DataFrame({"Latitude": lat, "SST": mean_sst})
15
16    return latitude_frame
```

We're just interested in *each* available latitude across all times and longitudes, so we just give placeholder values for maximum.

```
In [25]: 1 mean_by_lat = extract_mean_sst_by_lat(0, 500, 0, 500, surface_temperatures)
2 mean_by_lat.plot(x="Latitude", y="SST")
```

```
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x10b2caf28>
```



We will use `SciPy` to estimate a quick linear regression on these not-totally-linear data.

```
In [26]: 1 scipy_linreg = scipy.stats.linregress(x = mean_by_lat.Latitude, y = mean_by_lat.SST
2         scipy_linreg
```

```
Out[26]: LinregressResult(slope=-0.4352812287008252, intercept=305.1304619725177, rvalue=-0.9771853296131076, pvalue=5.3182628179698945e-61, stderr=0.010085152123907332)
```

Notice that `SciPy` is giving us R , which is correlation, and not R^2 . We can see that the correlation between latitude and SST is negative (increasing latitude - closer to the poles - results in lower mean SST across longitudes). We can calculate R^2 to get percentage of variation explained:

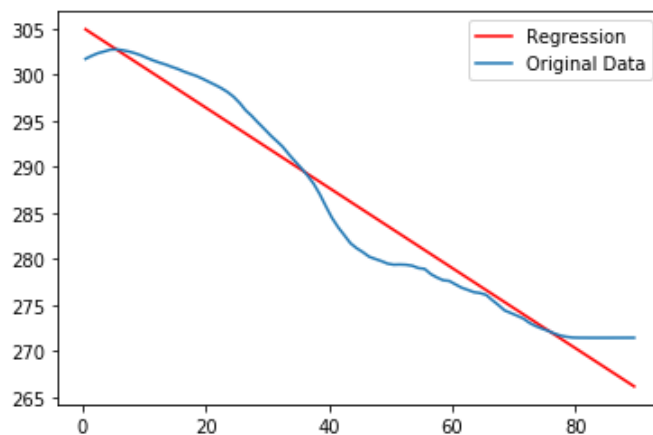
```
In [27]: 1 scipy_linreg.rvalue**2
```

```
Out[27]: 0.9548911684110777
```

So using this simple linear model, we can explain 95.4% of the variation in SST with latitude (*note*: correlation is not causation!).

```
In [28]: 1 mean_by_lat["Regression"] = scipy_linreg.slope * mean_by_lat.Latitude + scipy_linr
2
3 # Let's plot our data and regression using PyPlot
4 plt.plot(mean_by_lat.Latitude, mean_by_lat.Registration, color = "red", label = "Reg
5 plt.plot(mean_by_lat.Latitude, mean_by_lat.SST, label = "Original Data")
6 plt.legend()
```

```
Out[28]: <matplotlib.legend.Legend at 0x1a210520f0>
```



Part 2: Biological Data and FASTA files via SeqIO!

We'll do a short example of parsing and investigating data from the mouse genome (*Mus musculus*). This is just a collection of predicted genes from one location in the mouse genome that I have subsetting (so it's a small fraction of the overall data just for the first chromosome) for the purposes of this exercise.

```
In [29]: 1 mouse_genome = list(SeqIO.parse("data/Mus_musculus.GRCm39.cds.subsetted.fa", "fast
```

`SeqIO` allows us to treat every sequence in our FASTA file as an individual record and process it according to characteristics extracted from the FASTA format sequence file.

```
In [30]: 1 mouse_genome[0]
```

```
Out[30]: SeqRecord(seq=Seq('GGGACAGGGGGC', SingleLetterAlphabet()), id='ENSMUST00000178537.2', name='ENSMUST00000178537.2', description='ENSMUST00000178537.2 cds chromosome:GR  
Cm39:6:41510135:41510146:1 gene:ENSMUSG00000095668.2 gene_biotype:TR_D_gene transcript_biotype:TR_D_gene gene_symbol:Trbd1 description:T cell receptor beta, D region 1  
[Source:MGI Symbol;Acc:MGI:4439571]', dbxrefs=[])
```

The `SeqRecord` data type will always come with a field `seq` that is associated with every record inside of the FASTA file.

```
In [31]: 1 mouse_genome[0].seq
```

```
Out[31]: Seq('GGGACAGGGGGC', SingleLetterAlphabet())
```

We can perform typical array operations on these character array sequence entries.

```
In [32]: 1 len(mouse_genome[0].seq)
```

```
Out[32]: 12
```

Performing list operations on a character array

If we treat our string of mouse sequences as character arrays, we can perform *array operations* on those data.

```
In [33]: 1 sequence_as_list = list(mouse_genome[0])
```

Similarly, we can use **list comprehension** within Python to treat *all* our sequences as lists, and to process the data inside each list.

In bioinformatics, one metric we often use for a quick assessment of a sequence is **GC content**. You may remember from an introductory biology course that guanine and cytosine are a purine and a pyrimidine, respectively, that bind together within DNA (or RNA). When these two molecules latch together, they use 3 *hydrogen bonds*. This is in contrast to the bond between adenine and thymine (A and T, respectively), which only has two hydrogen bonds.

This small difference changes the melting temperature of DNA molecules that contain more C & G relative to the A & T that they contain. This is important for molecular reactions that we impose on DNA, and it also has pretty major implications for organisms...it's a common recommendation to make *taxonomic classifications* based on GC-content.

So, let's measure the GC content of some sequences!

```
In [34]: 1 GC_content_per_seq = [(list(curr).count("C") + list(curr).count("G"))/len(list(curr)) for curr in mouse_genome]
```

```
In [35]: 1 plt.hist(GC_content_per_seq)
          2 plt.xlim([0,1])
```

Out[35]: (0, 1)

