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

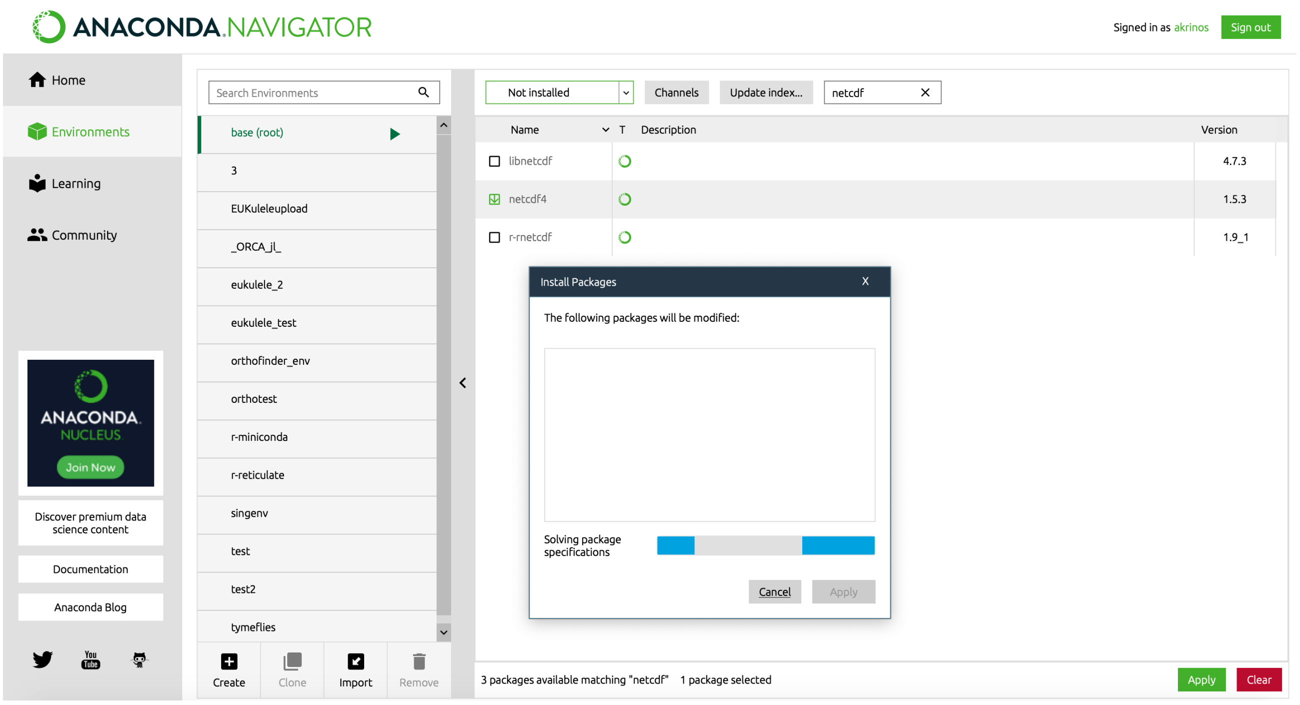
- Load netCDF files as well as a FASTA file for biological sequences
- Learn skills for plotting and analyzing data in 2D and 3D
- Linear and multinomial regression
- Statistical tests

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) 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.



Name	T	Description	Version
libnetcdf			4.7.3
netcdf4			1.5.3
r-rnetcdf			1.9_1

Install Packages

The following packages will be modified:

Solving package specifications

3 packages available matching "netcdf"   1 package selected

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 [ ]:
```python
import pandas as pd
import numpy as np
import datetime
from scipy import stats
from netCDF4 import Dataset
import matplotlib.pyplot as plt
```

In [ ]:
```python
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 *micro*meters 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 [ ]: 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 [ ]: surface_temperatures.ncattrs()
```

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

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

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

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

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 O1 (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 [ ]:
```python
surface_temperatures.variables
```

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

In [ ]:
```python
 surface_temperatures['lat'][:]
```

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 [ ]:
```python
sst = surface_temperatures['tos'][0,:,:]
```

In [ ]:
```
sst
```

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

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 [ ]:
```
sst = surface_temperatures['tos'][0,100:400,100:400]
plt.imshow(sst, interpolation='none')
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/) on whether this is a totally terrible colormap to use).

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

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 [ ]:
```
lon = surface_temperatures['lon'][100:]
lat = surface_temperatures['lat'][100:]
sst = surface_temperatures['tos'][0, 100:, 100:]

# We use vmin and vmax to set the bounds of the colorbar. Notice that these measur
plt.pcolor(lon, lat, sst, cmap=cmap, vmin=270, vmax=300)
```

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

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 [ ]:
```python
surface_temperatures["time"][4]

lon = np.subtract(surface_temperatures['lon'],180)
lat = surface_temperatures['lat']
sst = surface_temperatures['tos'][1,:,:]

fig, axs = plt.subplots(ncols=1, nrows=2, subplot_kw={'projection': "mollweide"})

pcm1 = axs[0].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27
axs[0].title.set_text('15 January')
fig.colorbar(pcm1, ax=axs[0])

lon = np.subtract(surface_temperatures['lon'],180)
lat = surface_temperatures['lat']
sst = surface_temperatures['tos'][20,:,:]

pcm2 = axs[1].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27
axs[1].title.set_text('18 August')
fig.colorbar(pcm2, ax=axs[1])
```

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

In [ ]:
```python
lon = surface_temperatures['lon'][100:]
lat = surface_temperatures['lat'][100:]

surface_temperatures["time"][4]

lon = np.subtract(surface_temperatures['lon'][100:],180)
lat = surface_temperatures['lat'][100:]
sst = surface_temperatures['tos'][1,100:,100:]

fig, axs = plt.subplots(ncols=1, nrows=2)

pcm1 = axs[0].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27
axs[0].title.set_text('15 January')
fig.colorbar(pcm1, ax=axs[0])

lon = np.subtract(surface_temperatures['lon'][100:],180)
lat = surface_temperatures['lat'][100:]
sst = surface_temperatures['tos'][4,100:,100:]

pcm2 = axs[1].pcolormesh(np.radians(lon), np.radians(lat), sst, cmap=cmap, vmin=27
axs[1].title.set_text('18 August')
fig.colorbar(pcm2, ax=axs[1])
```

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 [ ]:
```
## REGION 1
long_indices = np.where((surface_temperatures['lon'][:] > 290) & (surface_temperat
lon = np.subtract(surface_temperatures['lon'][long_indices],180)
lat_indices = np.where((surface_temperatures['lat'][:] > 30) & (surface_temperatur
lat = surface_temperatures['lat'][lat_indices]
sst = surface_temperatures['tos'][:,lat_indices[0],long_indices[0]]

# take mean over axis 1&2 (axis 0 is time)
mean_sst_region1 = np.mean(sst,(1, 2))

## REGION 2
long_indices = np.where((surface_temperatures['lon'][:] > 230) & (surface_temperat
lon = np.subtract(surface_temperatures['lon'][long_indices],180)
lat_indices = np.where((surface_temperatures['lat'][:] > 30) & (surface_temperatur
lat = surface_temperatures['lat'][lat_indices]
sst = surface_temperatures['tos'][:,lat_indices[0],long_indices[0]]

# take mean over axis 1&2 (axis 0 is time)
mean_sst_region2 = np.mean(sst,(1, 2))

## REGION 3
long_indices = np.where((surface_temperatures['lon'][:] > 320) & (surface_temperat
lon = np.subtract(surface_temperatures['lon'][long_indices],180)
lat_indices = np.where((surface_temperatures['lat'][:] > 60) & (surface_temperatur
lat = surface_temperatures['lat'][lat_indices]
sst = surface_temperatures['tos'][:,lat_indices[0],long_indices[0]]

# take mean over axis 1&2 (axis 0 is time)
mean_sst_region3 = np.mean(sst,(1, 2))
```

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

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

In [ ]:
```
date_start = datetime.datetime.strptime("1/1/2001", "%m/%d/%Y") # the start date i

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

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

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

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:

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{bmatrix} = \begin{bmatrix} 1 & a_0 \\ 1 & a_2 \\ \vdots & \vdots \\ 1 & a_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

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 $\mathbf{s} = \mathbf{Eb}$, 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:

$$\text{error}(\beta_0, \beta_1) = \sum_{t=1}^{n} (s_t - (\beta_0 + \beta_1 a_t))^2$$

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:

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

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 [ ]:
```
# A function which takes as input a set of latitude and longitude bounds
# and then calculates & returns mean SST across time and longitude.
def extract_mean_sst_by_lat(min_lat, max_lat, min_lon, max_lon, sst_frame):
    long_indices = np.where((sst_frame['lon'][:] > min_lon) & \
                            (sst_frame['lon'][:] < max_lon))
    lat_indices = np.where((sst_frame['lat'][:] >= min_lat) & \
                           (sst_frame['lat'][:] <= max_lat)) # use >= so that we c

    lon = np.subtract(sst_frame['lon'][long_indices],180)
    lat = sst_frame['lat'][lat_indices]
    sst = surface_temperatures['tos'][:,lat_indices[0],long_indices[0]]

    mean_sst = np.mean(sst,(0,2))
    latitude_frame = pd.DataFrame({"Latitude": lat, "SST": mean_sst})

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

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

In [ ]:
```
scipy_linreg = scipy.stats.linregress(x = mean_by_lat.Latitude, y = mean_by_lat.SS
scipy_linreg
```

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 [ ]:
```
scipy_linreg.rvalue**2
```

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

```
In [ ]:
mean_by_lat["Regression"] = scipy_linreg.slope * mean_by_lat.Latitude + scipy_linr

# Let's plot our data and regression using PyPlot
plt.plot(mean_by_lat.Latitude, mean_by_lat.Regression, color = "red", label = "Reg
plt.plot(mean_by_lat.Latitude, mean_by_lat.SST, label = "Original Data")
plt.legend()
```

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 subsetted (so it's a small fraction of the overall data just for the first chromosome) for the purposes of this exercise.

```
In [ ]:
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 [ ]:
mouse_genome[0]
```

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

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

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

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

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 [ ]:
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 [ ]:
```
GC_content_per_seq = [(list(curr).count("C") + list(curr).count("G"))/len(list(cur
```

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