Very brief introduction to the python libraries: xarray and dask

- `xarray` is similar to the `pandas` library, but for multi-dimensional data. Whereas `pandas` works well with 2-d or tabular data, it is common in oceanography (at least in physical oceanography) to have 3-d or 4-d data (with 3 spatial dimensions and 1 time dimension). `xarray` is also very useful for analyzing large volumes of data often in netcdf (.nc) format.
- `dask` is a library that parallelizes code (i.e. can run multiple computations at the same time) in a relatively easy and efficient way. It is very useful when using big data and integrates nicely with `xarray`.

This notebook is meant to only be a *very brief* introduction to these two libraries, so you know where to start if you'd like to use these tools in your own research/coding. There are a few very nice tutorials online about both of these libraries, including these two by Ryan Abernathey:

- "Xarray Fundamentals": https://rabernat.github.io/research_computing_2018/xarray.html (https://rabernat.github.io/research_computing_2018/xarray.html)
- "Dask for Parallel Computing and Big Data": https://rabernat.github.io/research_computing_2018/dask-for-parallel-computing-and-big-data.html# (https://rabernat.github.io/research_computing_2018/dask-for-parallel-computing-and-big-data.html#)

In this notebook, we will also download some free sea surface temperature (SST) data from NOAA using ftp (file transfer protocol) and the command 'wget'. This same method is used in the cartopy tutorial.

If you have not yet used the command 'wget', you will need to install it by typing `!conda install -c anaconda wget` in a new code cell (or in Anaconda Prompt).

Download the SST data

Uncomment the cell below to download mean SST data.

Note: you only need to run this once! Once you run the cell, you should see the data 'sst.mnmean.nc' appear in the same folder as this notebook is in. The data is ~57MB.

In [ ]:
```
#!wget ftp://ftp.cdc.noaa.gov/Datasets/noaa.oisst.v2/sst.mnmean.nc
```

xarray

First we need to import xarray and set our plots to show up in the notebook

In [ ]:
```
%matplotlib inline
import xarray as xr
```

Now let's open the data into an xarray dataset, which we will call 'ds' for short.

In [ ]:
```
ds = xr.open_dataset('../sst.mnmean.nc')
```

To view details of the dataset, simply type its name

In [ ]:
```
ds
```

So you can see that the data has a few things to note:

- the data is a `xarray.Dataset` typre
- 4 dimensions of different sizes: lat, lon, nbnds, time
- 3 coordinates: lat, lon, time
- 2 variables: sst and time_bnds
- a list of attributes (i.e. metadeta: telling us where and when the data is from, etc.)

You can start to understand just how useful xarray is with netcdf files, as it can load all different types of information about a dataset.

To access items within the dataset, we just type `ds.` followed by the aspect of the dataset we are interested in. The following few cells show a few examples.

See information about the variable 'sst':

In [ ]:
```
ds.sst
```

List information about the time dimension. Notice that the time interval is in months, and the data goes from 1981 to 2020.

In [ ]:
```
ds.time
```

List all dimensions:

In [ ]:
```
ds.dims
```

List all variables:

In [ ]:
```
list(ds.keys())
```

List all attributes:

In [ ]:
```
ds.attrs
```

One of the very nice features of xarray is the ease in which you can do simple data manipulation, such as taking the mean of a dataset. First we select the sst variable, and then we write out the function `.mean()` with the argument 'time' to take the time average.

In [ ]:
```
ds.sst.mean('time')
```

Another one of the great aspects of `xarray` is that it supports calling dimensions and variables by their names, instead of trying to remember which dimension is first or second, etc. (as we would need to do, for instance, with numpy).

Great, so we just took the mean of a dataset in one line! What if we want to plot the mean SST that we just calculated? It turns out that you can do this on the same line as well! All you have to do is add `.plot()` to the end!

In [ ]:
```
ds.sst.mean('time').plot()
```

This plot doesn't look very nice at the moment, but that's where the map plotting package cartopy comes to the rescue! If you want to try out cartopy, go over to the cartopy tutorial that Josué made for you!

Ok, that is all I'm going to do with xarray for now. I hope you can see how useful it can be, and you can check out the tutorial linked at the top of this notebook if you want to learn about xarray in more depth.

dask

In this brief dask tutorial (largely based on the one at https://tutorial.dask.org/00_overview.html (https://tutorial.dask.org/00_overview.html)), we will see how dask can help speed up your computations.

We will start by defining some very basic adding and multiplying functions that use the function `sleep()` from the `time` library. This sleep() function causes the function add() that we define below to pause for the number of seconds inside the parentheses."

In [ ]:
```
from time import sleep

# Define a function to add two numbers
def add(x,y):
    sleep(2) # pause for 2 seconds
    return x+y

# Define a function to multiply two numbers
def mlt(x,y):
    sleep(1) # pause for 1 second
    return x*y
```

In the next cell, we will use another 'magic' function similar to the `%matplotlib inline` function you have probably used numerous times. These function calls that begin with a `%` are called 'magic' functions in ipython notebooks. This time we will use `%%time` , which prints out the amount of time it takes to run all of the code in that particular cell.

If we call add() once and mlt() twice, can you guess how long it will take to run? It should take almost exactly 2 + 1 + 1 seconds.

In [ ]:
```
%%time

a = add(1,2)
b = mlt(1,2)
c = mlt(a,b)
```

On my computer, it took 4.01 seconds to run - pretty darn close to 4 seconds!

But, we could theoretically run all of these functions at the same time, since they are all independent calculations. `dask` can help us do that! We are now going to import a dask function called 'delayed'. It is so called because it doesn't run a function immediately, but stores the information to run a function until the user specifies a `.compute()` function, at which point the calculation is run, and in the most optimized way. Let's see an example:

In [ ]:
```
from dask import delayed
```

In [ ]:
```
%%time

a = delayed(add)(1,2)
b = delayed(mlt)(1,2)
c = delayed(mlt)(a,b)
```

Hmm, this claims that the calculations ran in 640 microseconds - but that doesn't make sense! Our two functions require at least a 1 second pause when run. The catch is that we haven't actually done the calculation yet. We have just created delayed objects that will run once we compute them.

In [ ]:
```
%%time

c.compute()
```

Now it looks like it only took 3.01 seconds to run the same calculation as before when it took 4 seconds! While 1 second doesn't sound like much, this can be scaled up with large amounts of data. You can save hours or even days or your time by parallelizing with dask!

Dask can do way more interesting things, but I'll leave you to explore it in more depth on your own - a good place to start is the tutorial linked at the top of this notebook.

Note that dask may not be necessary to use if you do not have big data! If you do have large amounts of data that take a long time to run, then dask is a great resource, and you can use it on your local computer or on computer clusters at unviersities or on the cloud!

In [ ]: