Dr. Arraylove, or How I learned to stop worrying and love xarray

I knew xarray was awesome, but it’s more awesome than I realized (especially for CMIP data)

1. Label-based indexing
2. Direct application of many math / plotting functions
3. Laziness
4. Plays nicely with Dask (still a pretty huge n00b on this one)

xarray was created to expand much of the pandas functionality to N-D arrays.

pandas: the Python version of R’s dataframe structures

- great for labeled data (regions, dates / times, model runs, etc)
- Label-based reshaping, indexing, slicing, and subsetting
- Easy to group, merge, and join

However, pandas has a big drawback when it comes to geosciences:

- pandas is a @$#%#$^ for n-d arrays!
- Enter…. xarray. Created by geoscientists for geoscientists*

*OK, lots of other people use it too, but not strictly true but geoscientists and their needs were integral for early xarray development

I already knew xarray was great for loading netCDFs (and other data formats, like GRIB).

- Better access to / use of netCDF metadata (all of the facts and information about your data, like how big it is and what the dimensions are, when it was created, etc.)
- **open_mfdataset** makes it easy to read many files at once (i.e., if each year is saved as a separate file)
- Plays nicely with **dask**, if you want or need to parallelize your code (run many things at once)

xarray is designed for use with labeled data

Think about a dataset you use in weather/climate analysis. It probably has dimensions that include *latitude, longitude,* and *time*. Depending on the variable and the data set, it might also include:

- *Vertical coordinate* (depth, height, pressure)
- *Ensemble number*
- *Region*
- *Model used* (if comparing across models)

xarray is designed for use with labeled data

xarray makes it easy to subset, index, etc along a specific dimension WITHOUT hard-coding your dimension OR writing extra lines of code

Code example: Extract latitudes 40-60N from surface air temperature (*tas*). Assume *tas* has dimensions of [*lon* x *lat* x *time*].

Without *xarray*:

```
print(np.where(tas.shape == len(lat))) #or something similar; shows where to slice

tas_slice_ind = np.isin((lat,np.arange(40,60)))

tas_slice = tas[:,tas_slice_ind,:]
```

With *xarray*:

```
tas_slice = tas.sel(‘lat’=slice(40,60))
```

xarray is designed for use with labeled data

- `slice` is for selecting a range along a labeled dimension
- 4 different ways to select one specific index:
  - `sel`: named dimensions, labeled indices (`tas.sel(‘time’=’2001-01-10’)`); pick out the `tas` data for Jan 10, 2001
  - `isel`: named dimensions, positional indices (tas.sel(‘time’=10)); pick out the tas data for the 10th time step
  - `loc`: positional dimensions, labeled indices. (tas.loc[:,:,`’2001-01-10’]`); find `tas` data for Jan 10, 2001; but we have to know which dimension corresponds to `time`
  - `iloc`: positional dimensions, positional indices. (tas.iloc[:,:,`10]`); select `tas` data for 10th time step, but we have to know which dimension corresponds to `time`

xarray is designed for use with labeled data, and directly applies math and plotting functions to those data.

In many other environments (Numpy, MATLAB), you would generally identify your dimensions ahead of time (look at the metadata, identify based on the size of your dimensions, etc), and write your code with that in mind.

- For datasets like CMIP, or other multi-model ensembles, different modeling centers don’t necessarily use the same ordering (e.g., [*lat* x *lon* x *time*] vs [*time* x *lat* x *lon*])
- So if you’d want to take a zonal mean, in MATLAB or Numpy, you’d figure out which dimension corresponded to *lon*, and take a mean along that dimension
- In xarray, you can just take the mean along “lon”--you can use the NAME of the dimension, and xarray will figure out where that is

xarray is designed for use with labeled data, and directly applies math and plotting functions to those data.

Code example: let’s say you want to take a zonal mean of surface air temperature (*tas*). Assume *tas* has dimensions of [*lon* x *lat* x *time*].

Without *xarray*:

```
dim_to_average = np.where(tas.shape == len(lon))

tas_mean = np.nanmean(tas,axis=dim_to_average)
```

OR: identify which dimension *lon* is ahead of time, and hard-code it (and re-write your code any time you use it on data with a different order)

```
tas_mean = np.nanmean(tas,axis=0)
```

With *xarray*:

```
tas_mean = tas.mean(‘lon’)
```

xarray is designed for use with labeled data, and directly applies math and plotting functions to those data.

Since xarray is designed for use with labeled data, it means:

- Fewer lines of code
- No need to hard-code or rewrite code when doing same analysis on slightly different datasets
- Easier to index and subset data
- Direct application of math and plotting functions included in `Numpy`, `Scipy`, and `matplotlib`

This last one was extra-exciting for me, because it means...way less unnecessary conversion from `Numpy`!! (note that functions not included in `Numpy`, `Scipy`, and `matplotlib` do not have this functionality)

xarray is lazy, which means your code runs faster

What is **lazy computing? Lazy computing** means that you don’t actually do anything until you need a value of some kind:

- `print` statement
- `plot` statement

So you can do several math operations without actually loading the data (xarray remembers the commands, and doesn’t apply them until you want to make a plot or print something) (you DO have to load the data eventually

xarray is lazy, dask is lazier, so this is even more prominent when you’re using dask

Full example comparing `xarray` and `netCDF4` can be found on github

Link: https://github.com/marie-mcgraw/teaching_examples/tree/master/xarray_examples

If you can’t view the notebook natively, you can try https://nbviewer.jupyter.org/

Other fun considerations

In academic research, our main goal is (usually) a scientific question--code is just a tool we use to probe our questions. Our job is to produce new scientific results, not to write really amazing software.

Many of us are partially or entirely self-taught, learning to code as we go in the service of doing some other task.

Most of our code falls under the category of **research code** (vs **production code**, such as a software engineer might write)

Other fun considerations

Research Code

- Designed to be exploratory--concerned with testing new ideas
- Typically only used by an individual or small group of people
- Not especially concerned with stability or versatility--we write code to address OUR specific problems, and we don't spend a lot of time thinking about other uses or other ways of breaking it
- We don't care (that much) about speed
- Little or no maintenance

Production Code

- Designed to work right every time (ideally)
- Created and used by a large number of people
- Stability matters--need to be able to anticipate how people might break it, all the ways in which they might use your software, etc
- Needs to be reliable, maintained, and reasonably fast

Most considerations of production-quality code are probably not worth it for those of us in academia.

- We don’t have enough training to write production quality code--figuring out how to do this is hard (**especially without expert guidance**), and takes time away from our primary job

- Do we find all of the extra hurdles between research code and production code interesting and fun? Probably many of us don’t (otherwise we’d be making bank at Amazon)

- Software doesn’t pay our bills, or directly help us get jobs...for now

However, as the world evolves, there are some things worth considering

- Reproducibility requirements are being taken more seriously
- CS-intensive analysis techniques are gaining lots of steam in geosciences (although this doesn’t necessarily help you write production-quality code)
- More people who DO have legitimate CS training and backgrounds are interested in applying their skills to geoscience problems (see workshops like Climate Informatics)--maybe they can teach us? (this seems dumb, but I got much better at things like I/O, writing intermediate data, and using useful error and print statements from working with CS people)
- More of us are ultimately considering (and ending up in) jobs that require us to write higher-quality code, whether or not they’re inside geosciences or not. Even if we don’t end up applying most of these ideas while we’re in academia, it’s helpful to know what’s out there!

A few things I’m trying to get better at, going forward

- **Documentation:** I think I’m usually okay about commenting my code, but I would like to get better at writing README files, so that other people (including FutureMarie) can use it more easily
- **Keep scripts shorter:** It’s easier, better, and often more reproducible to have more, shorter scripts that do specific tasks (it’s easier to edit and understand your code, too)
- **Think harder about where to break up scripts:** When should I end one script and start a new one? Should I perform all of these calculations together or spread them out over multiple scripts? How do these scripts interact with each other?
- **Write functions for things I do all the time,** instead of just being lazy and using copy/paste all the time
- **Properly learn how to create classes**: I...do not properly understand this