Section 0: The Setup

For this class, you should already know about:

- Git and Sourcetree
- Installing stuff through pycharm
- Numpy

Conda Install Things

Make sure you have installed the following packages:

- numpy
- pandas
- jupyter
- statsmodels (we might not use this)

To do this, open Pycharm, open the terminal from within Pycharm, and then use the following command:

```
conda install numpy pandas jupyter statsmodels
```

Note: `iPython` is needed for `jupyter`, so when you install `jupyter`, the `iPython` dependency is also installed, as is a bunch of other things.

iPython

iPython is interactive Python. Python shell is already *interactive*, but iPython provides really nice features for a truly interactive experience. Some of these featuers are:

- Sessions are stored (like in R)
- Command line commands work
- Tab completion
- Cells

Jupyter

Jupyter is iPython in a web browser with a few other nice features. One of the main features of Jupyter is that you can run the notebook on one server, and use it from another. At IHME, people typically run notebooks on the cluster so they can get access to a ton of memory and computer power, and access the notebook locally through a browser.

Setting up a jupyter notebook on the cluster is on the Hub and won't be covered. We'll be running notebook's locally.

Local setup

Linux or Unix

Run these commands:

```
cd ~  # change directory to home.
mkdir .jupyter  # make a directory named .jupyter
touch .jupyter/jupyter_notebook_config.py  # make an empty file in the directory
you just made.
mkdir notebooks  # we'll be using this directory to store notebooks you run.
```

Then find an open the `.jupyter/jupyter_notebook_config.py` file and put this stuff in it:

```
c.NotebookApp.ip = "*"
c.NotebookApp.port = 8888
c.NotebookApp.notebook_dir = "~/notebooks"
c.NotebookApp.open_browser = False
c.NotebookNotary.db_file = ":memory:"
```

Now get the training materials. To do this, change directory into the `notebooks` directory and then clone the repository.

```
cd ~/notebooks
git clone
https://tangkend@stash.ihme.washington.edu/scm/~tangkend/pandas_training.git
```

Now run from the command line `jupyter notebook`. Once it is running, open your favorite browser and go to `localhost:8888` and sign in.

Windows

If you use `GitBash` then just use

```
cd <where-ever-you-want>
git clone
https://tangkend@stash.ihme.washington.edu/scm/~tangkend/pandas_training.git
```

Otherwise, you can use SourceTree. Open up SourceTree and clone

```
https://tangkend@stash.ihme.washington.edu/scm/~tangkend/pandas_training.git
```

The destination can be anywhere you'd like.

Now start Jupyter Notebook in the directory you just cloned the training materials into. To do that,

open command prompt, change directories into the cloned directory, and then use `jupyter notebook` and hopefully that works. If it doesn't, uninstall windows and install linux.

Ok if it doesn't work, maybe one of the TAs can help, or we can try reading 3.1.1 of http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html

One New Feature

Jupyter notebooks have a lot of the same features as ipython notebooks. One new feature is the `shift+tab` documentation.

Caveats

Jupyter notebooks is used for exploration and demonstration. There are several pitfalls that can cause problems, some of which are:

1. order of cell execution is critically important.
2. cells are not guaranteed to be idempotent. That means just because it worked once, running the same cell again might not produce the same results. That's because of point 1 (order of execution is critically important).
3. variables have a really wide scope.

To alleviate some of the finicky parts of jupyter notebook, try Kendrick's suggestions:

1. Don't reuse varaible names. This is helpful no matter how you're writing code.
2. Stick to using Jupyter notebooks for exploration. Use pycharm or your preferred development environment to develop and test your code.
3. Don't send your coworkers jupyter notebooks as runnable code. Develop it in pycharm, write tests, and then send it.

Section 1: Pandas

We will not cover all of Pandas in this course. Please refer to the documentation when you're working.

- Homepage: http://pandas.pydata.org
- Documentation: http://pandas.pydata.org/pandas-docs/stable

Section 2: Pandas Series

Goal: be comfortable exploring an API, and understanding alignment on an index.

Pandas Series are a lot like Numpy Arrays, so they should feel familiar.

If you ever find yourself with numpy arrays and pandas series, you can perform operations between them and the numpy array will be **upcast** into a Pandas Series.

Indexing

In Numpy Arrays, there was only positional indexing with integers. In Pandas Series, we can use customized indexing which is more flexible.

Operations between Series will be automatically aligned by their index. This can cause problems if you aren't careful, but more often than not this is exactly the behaviour you want.

Large API

An API is an application programming interface. That's just fancy lingo for "the built-in things you can do". Pandas Series has lots of built-in functions so you don't have to reinvent the wheel. Some commonly used functions:

- mean
- max
- median

There are SO MANY different functions. We won't cover any of them here, so refer to the Pandas Docs.

Section 3: Intro to Pandas DataFrame

Goal: be able to describe a dataframe and do simple subsetting and operations on it.

Constructing a DataFrame from Series

Like with everything Pandas, when constructing a DataFrame from Series everything is aligned by its index.

In the demo, the following functions will be shown:

- `reset_index`
- `rename`
- `sort_values`

A more realistic scenario

It is more likely that you'll be given a data file and asked to do something with it. Before you can do anything useful, it is important to understand what the data looks like.

Reading in a CSV

The command is `pd.read_csv("data.csv")`. There are other file formats, and lots of fancy things you can do with `read_csv` but we won't cover that now.

Initial exploration

Pandas DataFrames provides lots of wonderful built-in functions for exploring the data in a DataFrame. Some common ones are:

- head
- shape
- columns
- unique
- mean/var
- max/min
- describe
- percentile

Hands-on-part:

Take some time to explore the data in Jupyter Notebook. Answer the following questions:

- Over what time period does this dataset have data for?
- Over the entire dataset, what is the average mortality rate?
- What is the maximum and minimum populations?

Selecting data

Data files will usually contain more data than you need for your assigned task. To select columns, use the square bracket notation `dataframe[columns]`. To select rows, you can use `dataframe.query()`

You can get column by accessing it as if it were a `member` of the dataframe like `dataframe.mortality_rates`, but this is bad practice. For example, the `mean` column would not be accessible this way because `dataframe.mean` would always return the mean function.

Hands on part

- For males in age group 12 and year 2016, what locations have the highest mortality rates? (highest 12)
- Which locations have the smallest female age group 19 population in year 1991? (smallest 14)

Adding new data (columns)

Sometimes, the data you have needs an additional column that you can obtain by performing operations on the existing columns. You can easily assign new columns to a dataframe:

1. `dataframe["is_true"] = True`
2. `dataframe["a_times_b"] = dataframe["a"] * dataframe["b"]`

In the first example, the scalar value `True` is **broadcasted** to all of the rows. In the second example, the data produced by the product is aligned by its index and assigned to the `a_times_b` column.

Hands-on-part:

Take some time to explore the data in Jupyter Notebook. Answer the following questions:

1. For America (location id 102) in 2016, did more males die, or did more females? How many?

Section 4: Indexing

Goal:

- setting an index and a multi index
- slicing into an index using .loc and .iloc
- know how to use boolean indexing
- know the difference between a copy and a view

This is a more content-dense section that the previous ones so get ready.

Setting an index

The default index is the same boring 0-to-N that we've seen with Numpy and Series, but like Series we can apply a more meaningful index to DataFrames.

Use `data.set_index(["location_id", "age_group_id", "sex_id", "year_id])` to set a multi-index. If the data isn't already in the right order, you'll need to use `sort_index` to get it in the right order.

It looks different, but why is this useful?

Accessing data via the index

Before, we showed you `data.query()`. If we have an **index** or a **multi-index** applied to the data, then we can use that to easily access specific parts of the data using `.loc`.

You can do more complicated slicing with `pd.IndexSlice`. I don't think I'm going to cover this, but maybe you'll remember that it exists and look it up when you need it.

We can also use `.iloc` which accesses the index as if it were using the 0-to-N index, but I've never found this useful.

Boolean indexing

Boolean data

If we try to evaluate a truth statement with a dataframe, the result is a dataframe of Trues and Falses. If we evaluate a truth statement with a series, we'll get a series of Trues and Falses. We can use this boolean series to index into the data.

Indexing with the boolean series

You can use a boolean series with `.loc` to get a **view** of a subset of the original dataframe. More on this later.

You can also use a boolean series with just plain old square brackets `dataframe[boolean_series]`. This returns a **copy**. I recommend not using plain old square brackets except for accessing columns, but I don't have any evidence for why this would be bad to do.

Views and Copies?

You may have noticed that the boolean indexing is really similar to `.query`. This is true. The main difference is that `query` returns a **copy** and `.iloc[boolean_series]` returns a **view**.

With a copy, no matter what you do with it, you will never modify the original dataframe. This is useful!

With a view, the change will update the original. This is also useful!

Knowing when to use a **view** and when to use a **copy** will help you do your work. Knowing when you are using a **view** or a **copy** will help you produce less bugs.

Section 5: Reshaping data

Goal: be able to use data from two different sources to produce a result.

So far, we've covered some basic operations to compute new results and produce some useful statistics from data. Time to throw a wrench in the operations.

Hypothetical scenario: the data you get isn't already perfectly formatted. So far we've been working with one dataframe that has both mortality rates and population in it. What if they were two different datasets?

We're going to cover two sets of techniques for reshaping data:

1. pivot and melt
2. stack and unstack

Combining Data

Now that the two data sets match, how do we combine them? There are three methods:

1. concat
2. merge
3. join

Section 6: Aggregating and transforming data

Goal: compute global all-cause all-age both-sex aggregate for gbd year ids.

A common question is: how much does each family weigh? We can answer this with pandas **groupby**.

A more realistic question is: in 2010, how many male deaths were there in countries more populous than Japan? We can also answer this with **groupby**.

Other things if we have time

A list of a few extra things:

- aggregation `agg`
- transform
- apply

Section 7: CSVs and HDFs

We probably won't have time for this.

If we do I might cover:

- datetime formatting
- floating point precision (or imprecision) in CSVs
- HDF and the table vs fixed formats