

3803ICT

Data Analytics

**Lab 02 –** **Data preparation**

**Trimester 1 - 2018**

Table of Contents

[I. Pandas 3](#_Toc507059141)

[1. Introducing pandas 3](#_Toc507059142)

[2. Series 6](#_Toc507059143)

[3. DataFrames 8](#_Toc507059144)

[4. Missing Data 13](#_Toc507059145)

[5. GroupBy 14](#_Toc507059146)

[6. Merging, Joining and Concatenating 16](#_Toc507059147)

[7. Operations 19](#_Toc507059148)

[8. Data Input and Output 22](#_Toc507059149)

[II. Data Nornalization 23](#_Toc507059150)

## **Pandas**

### Introducing pandas

Pandas is a Python library that makes handling tabular data easier. Since we're doing data science - this is something we'll use from time to time!

It's one of three libraries you'll encounter repeatedly in the field of data science:

**Pandas**

Introduces "Data Frames" and "Series" that allow you to slice and dice rows and columns of information.

**NumPy**

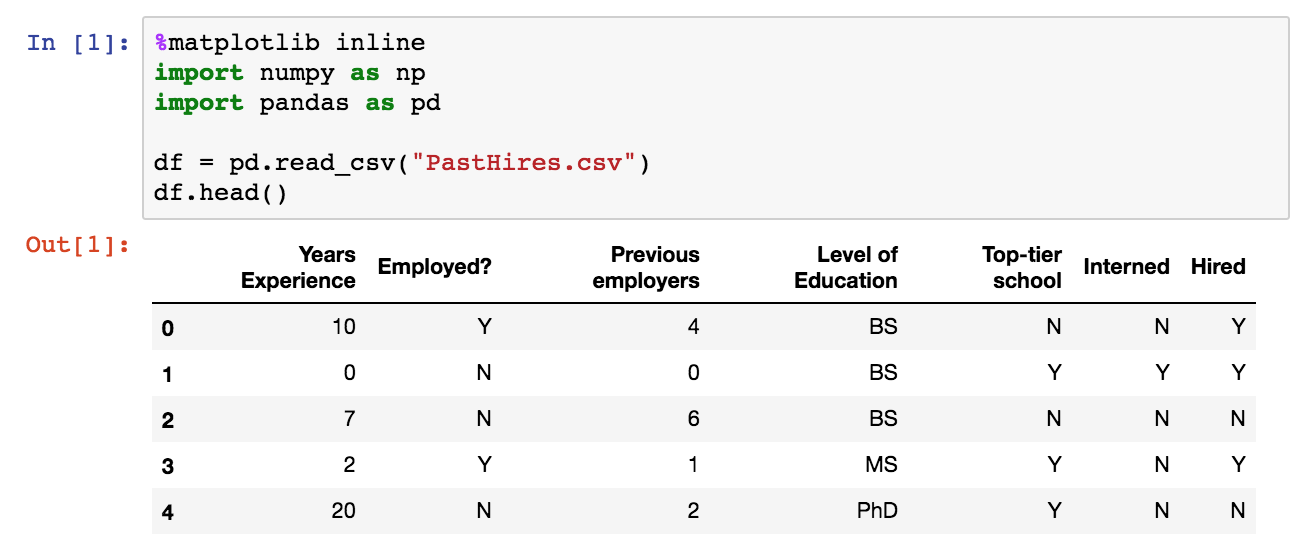
Usually you'll encounter "NumPy arrays", which are multi-dimensional array objects. It is easy to create a Pandas DataFrame from a NumPy array, and Pandas DataFrames can be cast as NumPy arrays. NumPy arrays are mainly important because of...

**Scikit\_Learn**

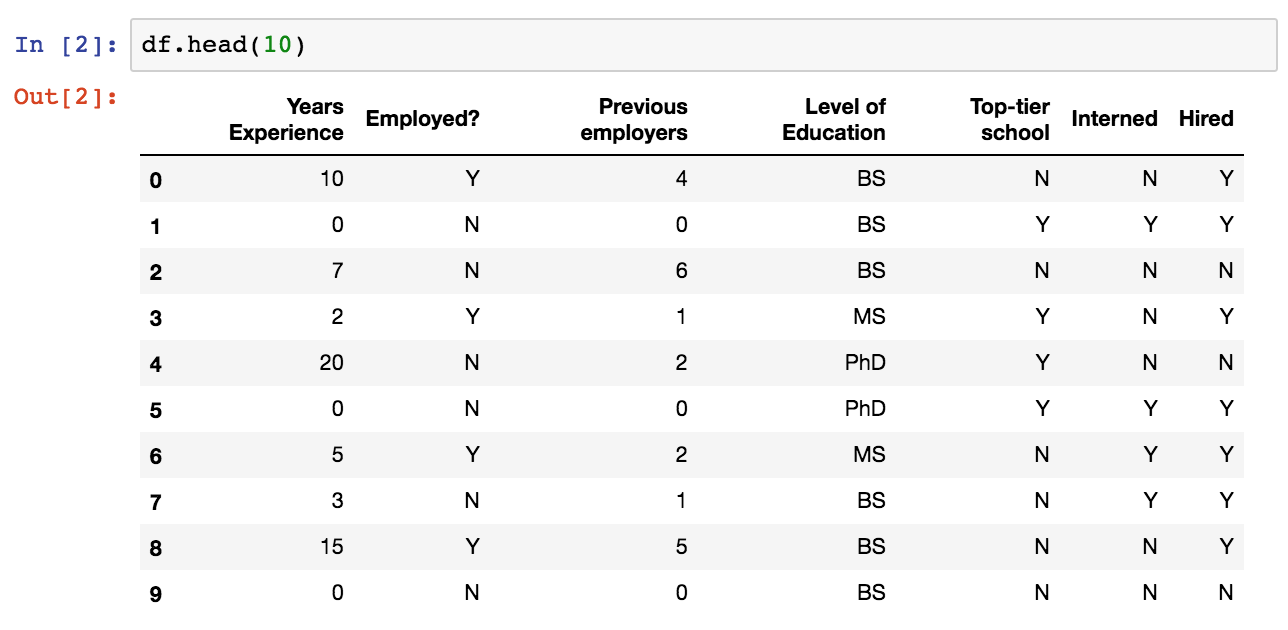
The machine learning library we'll use throughout this course is scikit\_learn, or sklearn, and it generally takes NumPy arrays as its input.

So, a typical thing to do is to load, clean, and manipulate your input data using Pandas. Then convert your Pandas DataFrame into a NumPy array as it's being passed into some Scikit\_Learn function. That conversion can often happen automatically.

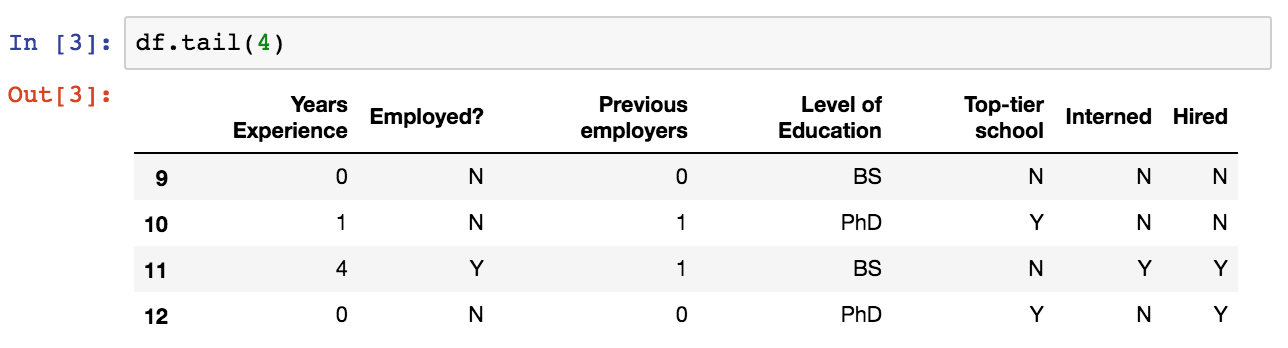
Let's start by loading some comma-separated value data using Pandas into a DataFrame:



head() is a handy way to visualize what you've loaded. You can pass it an integer to see some specific number of rows at the beginning of your DataFrame:



You can also view the end of your data with tail():



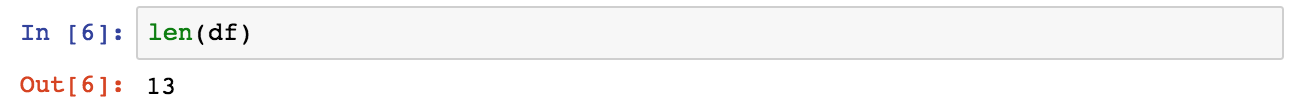
We often talk about the "shape" of your DataFrame. This is just its dimensions. This particular CSV file has 13 rows with 7 columns per row:



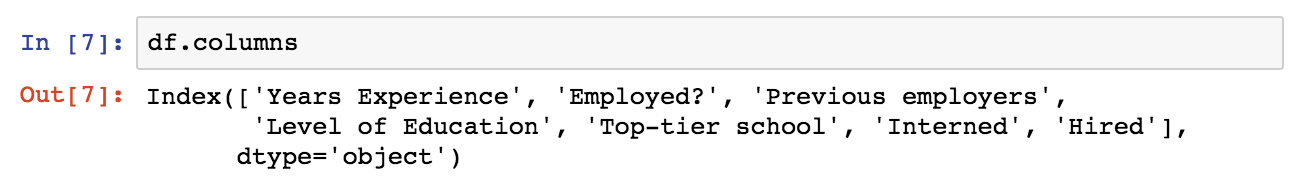
The total size of the data frame is the rows \* columns:



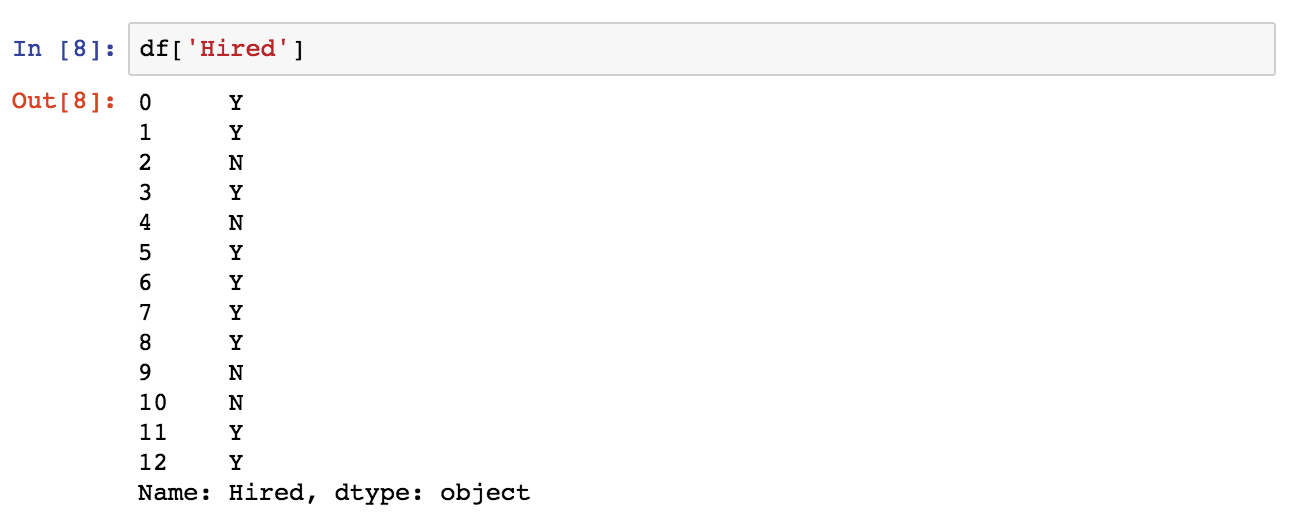
The len() function gives you the number of rows in a DataFrame:



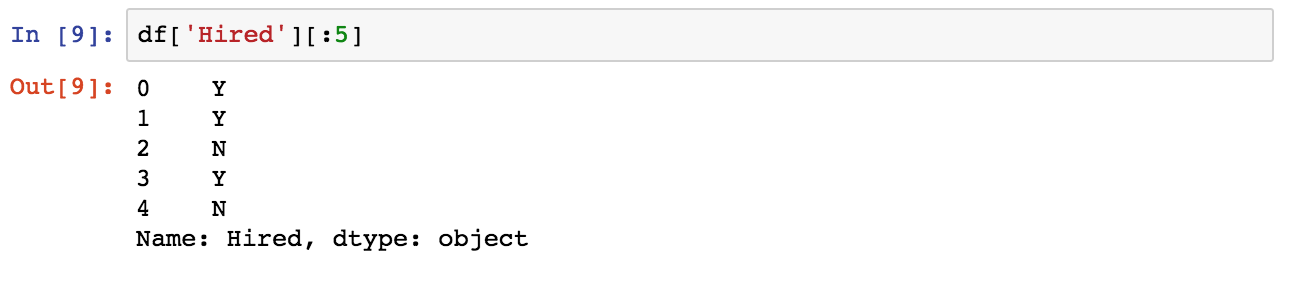
If your DataFrame has named columns (in our case, extracted automatically from the first row of a .csv file,) you can get an array of them back:



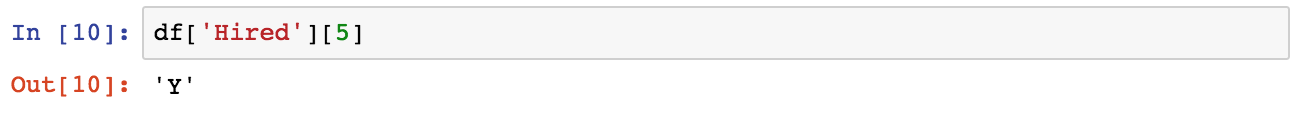
Extracting a single column from your DataFrame looks like this - this gives you back a "Series" in Pandas:



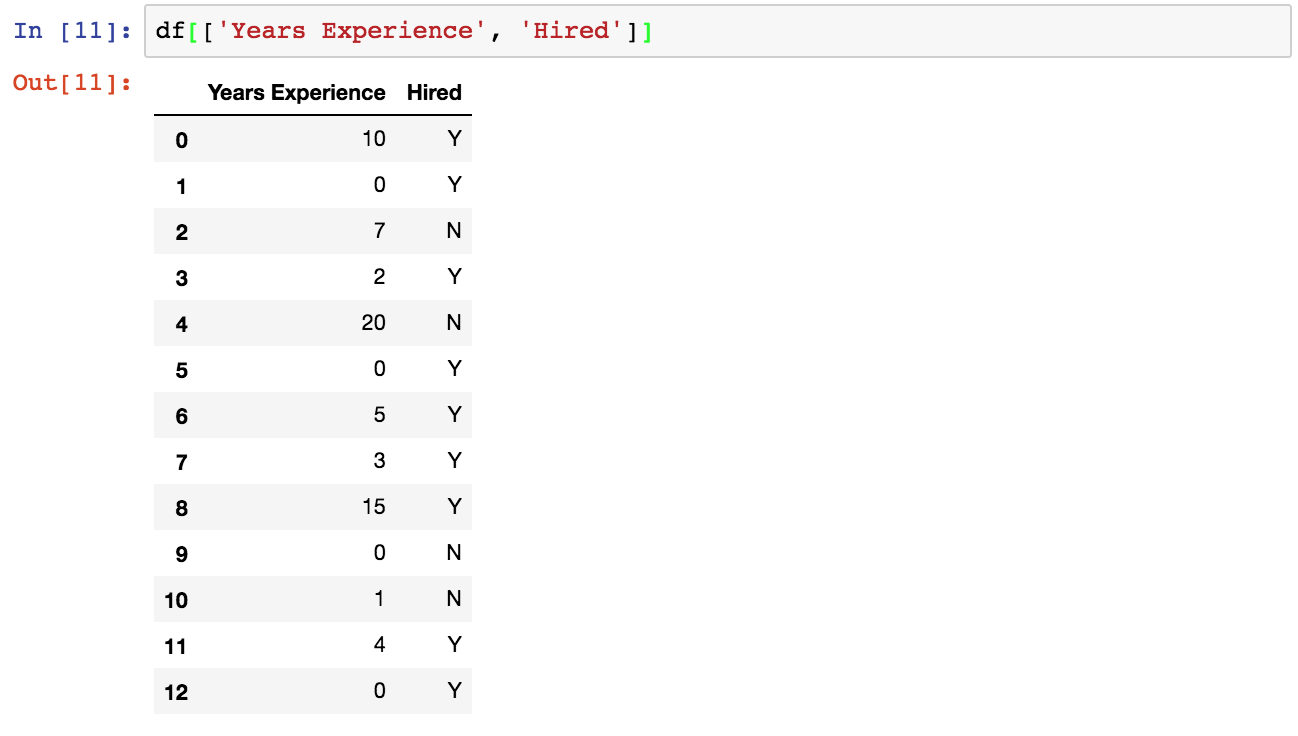
You can also extract a given range of rows from a named column, like so:



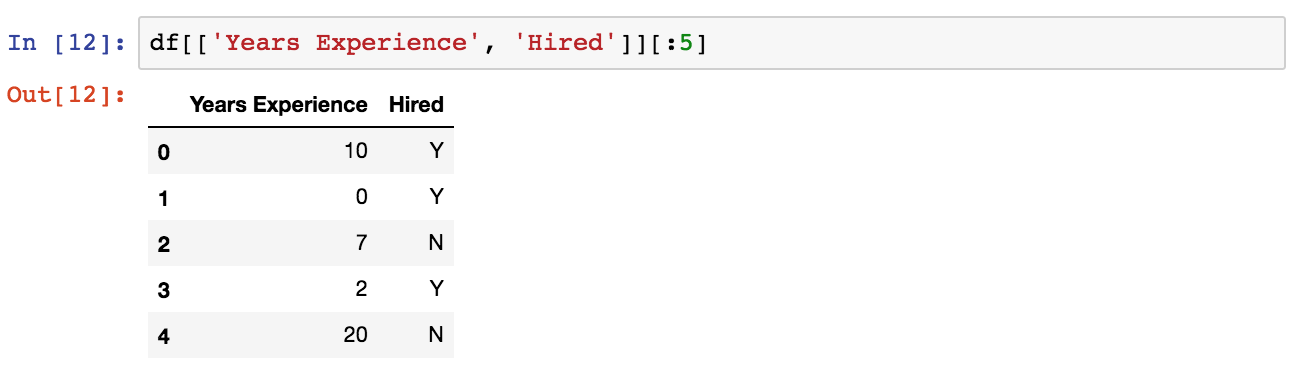
Or even extract a single value from a specified column / row combination:



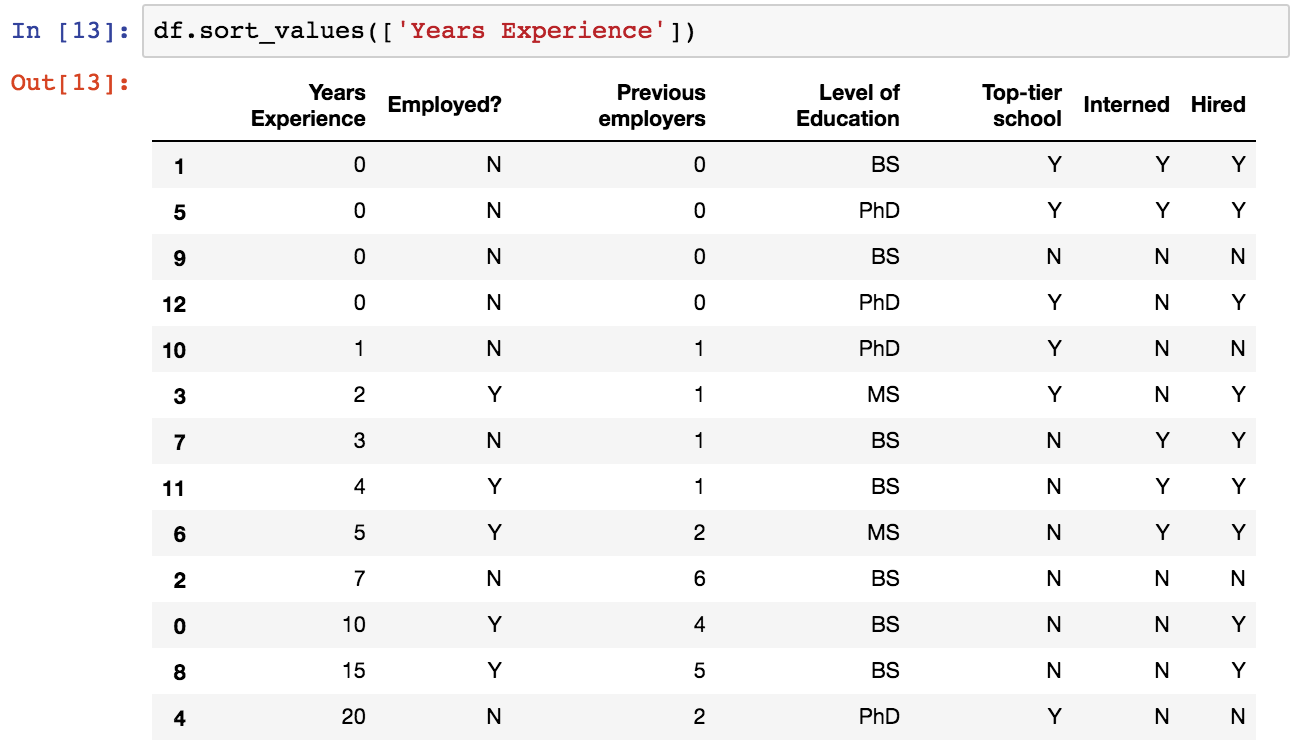
To extract more than one column, you pass in a list of column names instead of a single one:



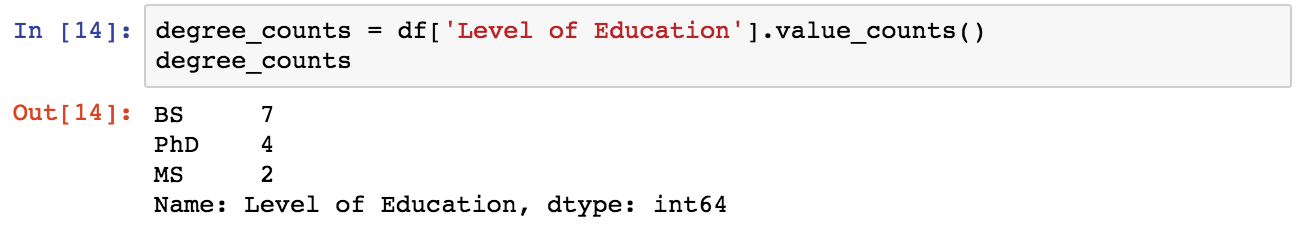
You can also extract specific ranges of rows from more than one column, in the way you'd expect:



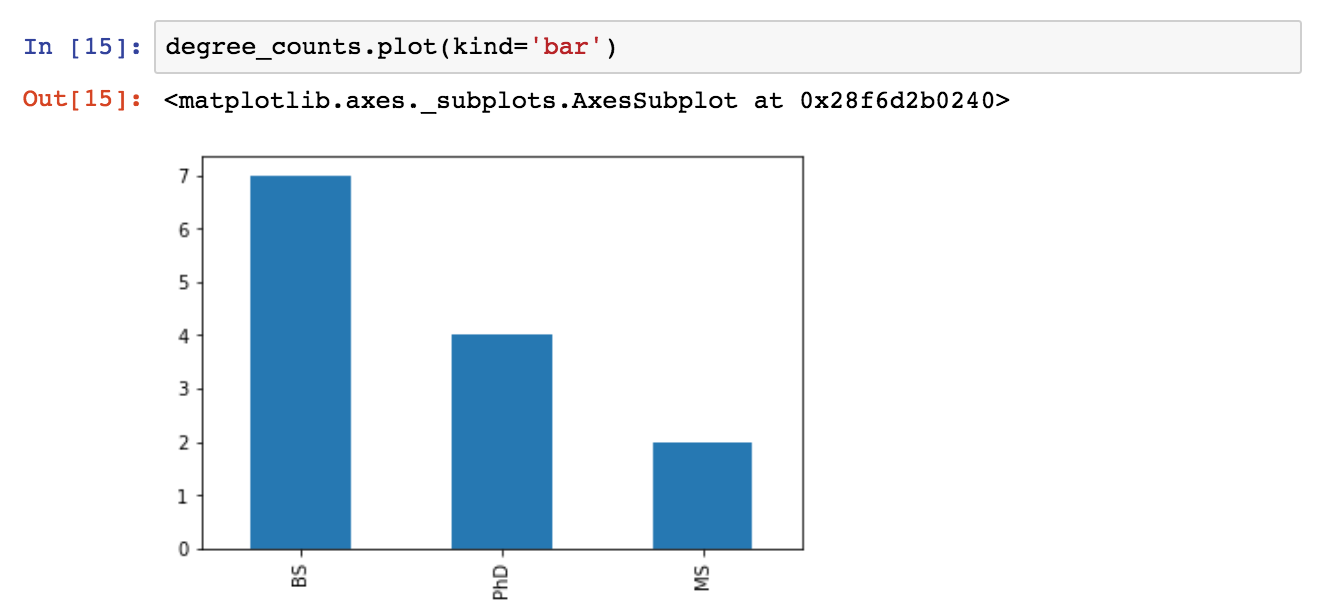
Sorting your DataFrame by a specific column looks like this:



You can break down the number of unique values in a given column into a Series using value\_counts() - this is a good way to understand the distribution of your data:



Pandas even makes it easy to plot a Series or DataFrame - just call plot():

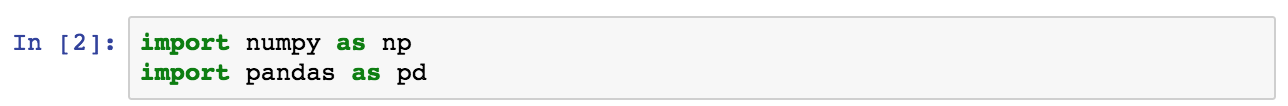


### Series

The first main data type we will learn about for pandas is the Series data type. Let's import Pandas and explore the Series object.

A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location. It also doesn't need to hold numeric data, it can hold any arbitrary Python Object.

Let's explore this concept through some examples:

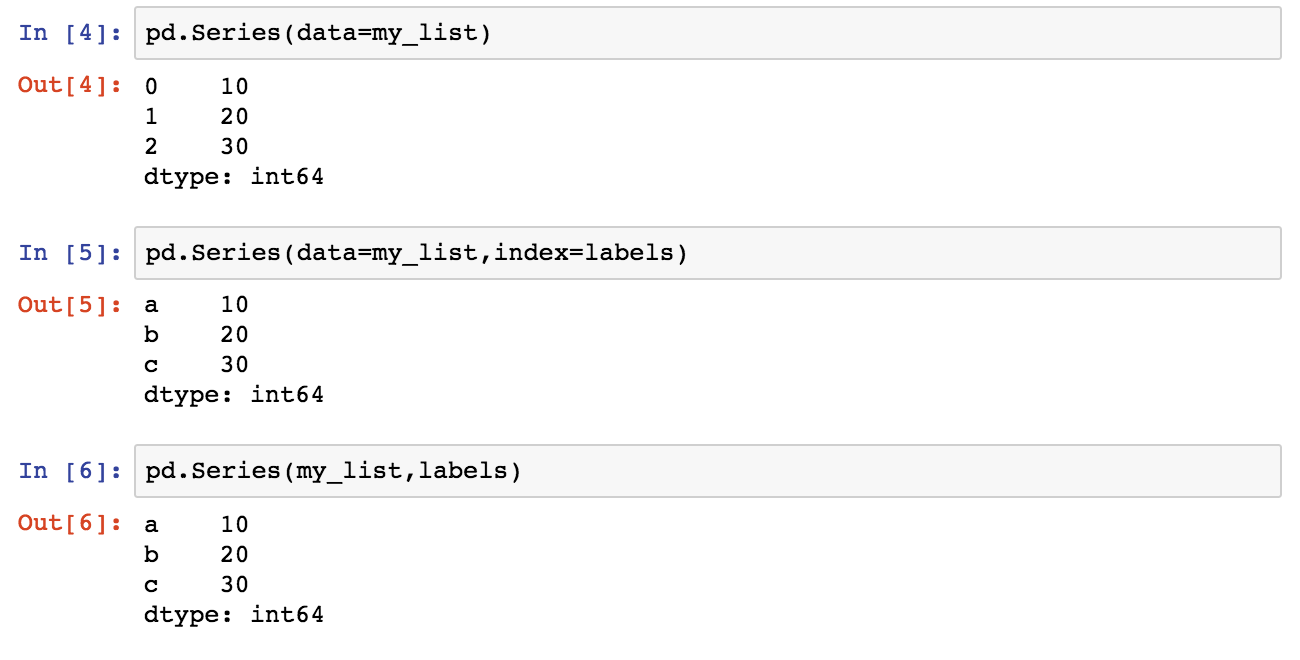


* 1. ***Creating a Series***

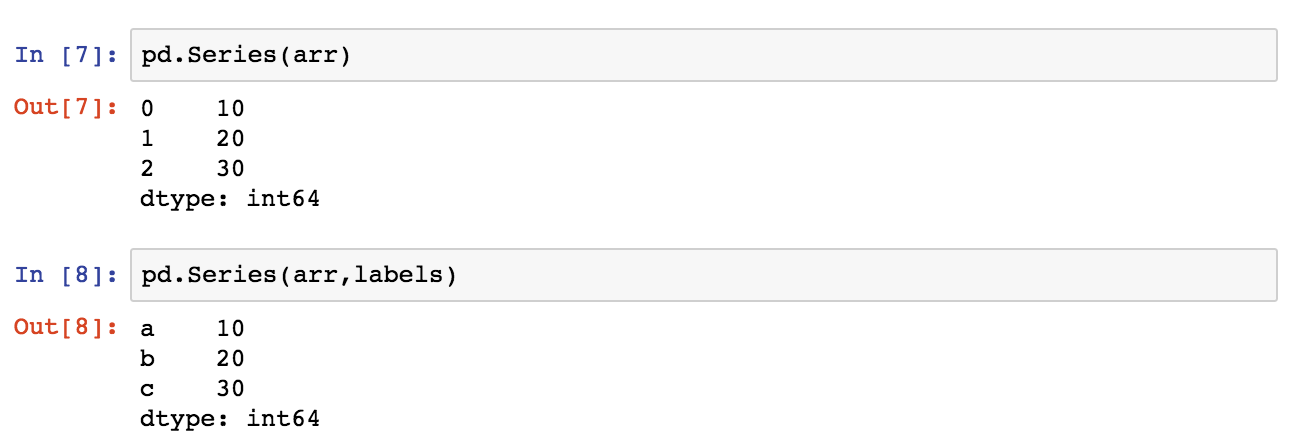
You can convert a list,numpy array, or dictionary to a Series:



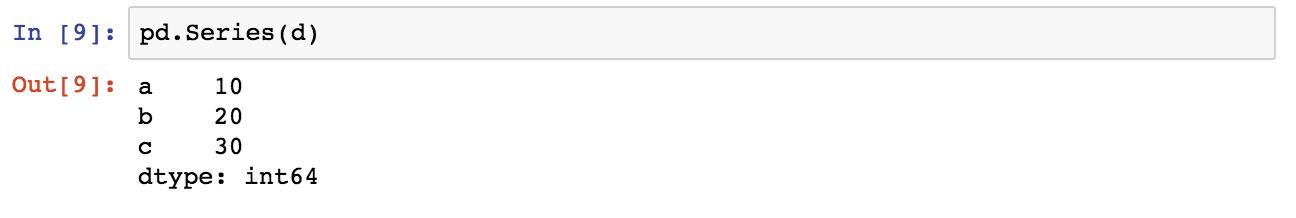
Using Lists



NumPy Arrays

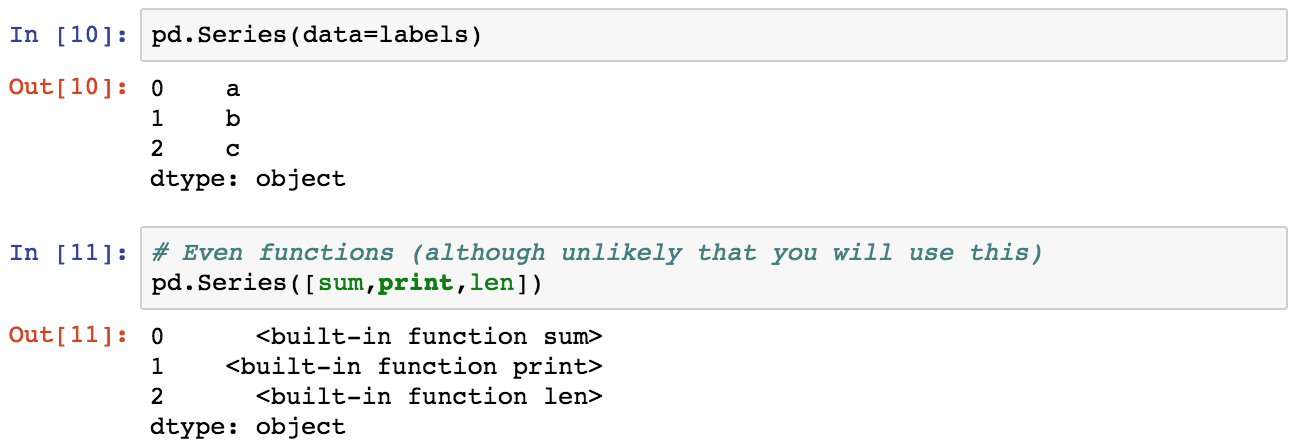


Dictionary



* 1. ***Data in Series***

A pandas Series can hold a variety of object types:

******

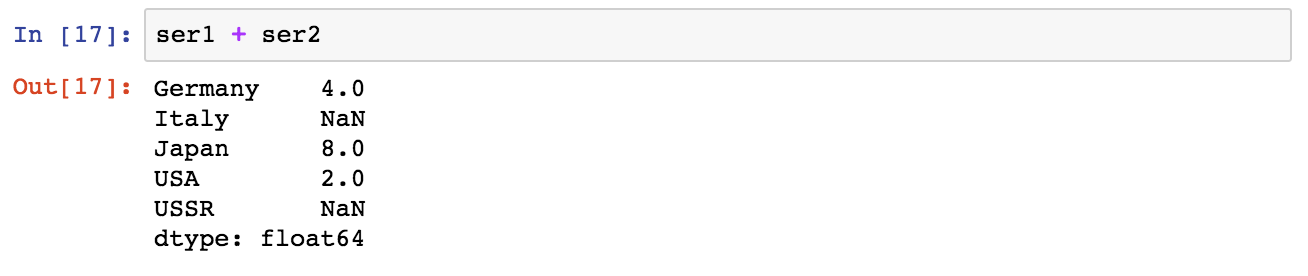
* 1. ***Using an index***

The key to using a Series is understanding its index. Pandas makes use of these index names or numbers by allowing for fast look ups of information (works like a hash table or dictionary).

Let's see some examples of how to grab information from a Series. Let us create two sereis, ser1 and ser2:

******

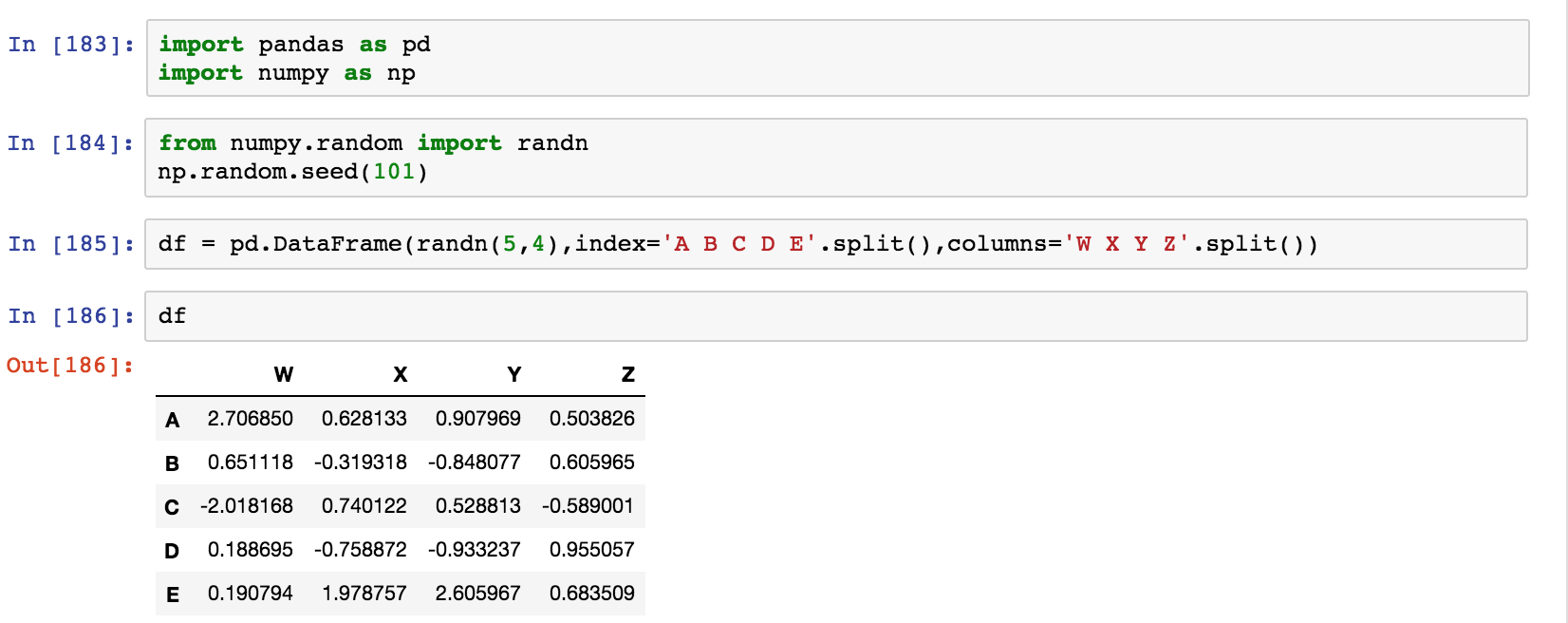
Operations are then also done based off of index:



Let's stop here for now and move on to DataFrames, which will expand on the concept of Series!

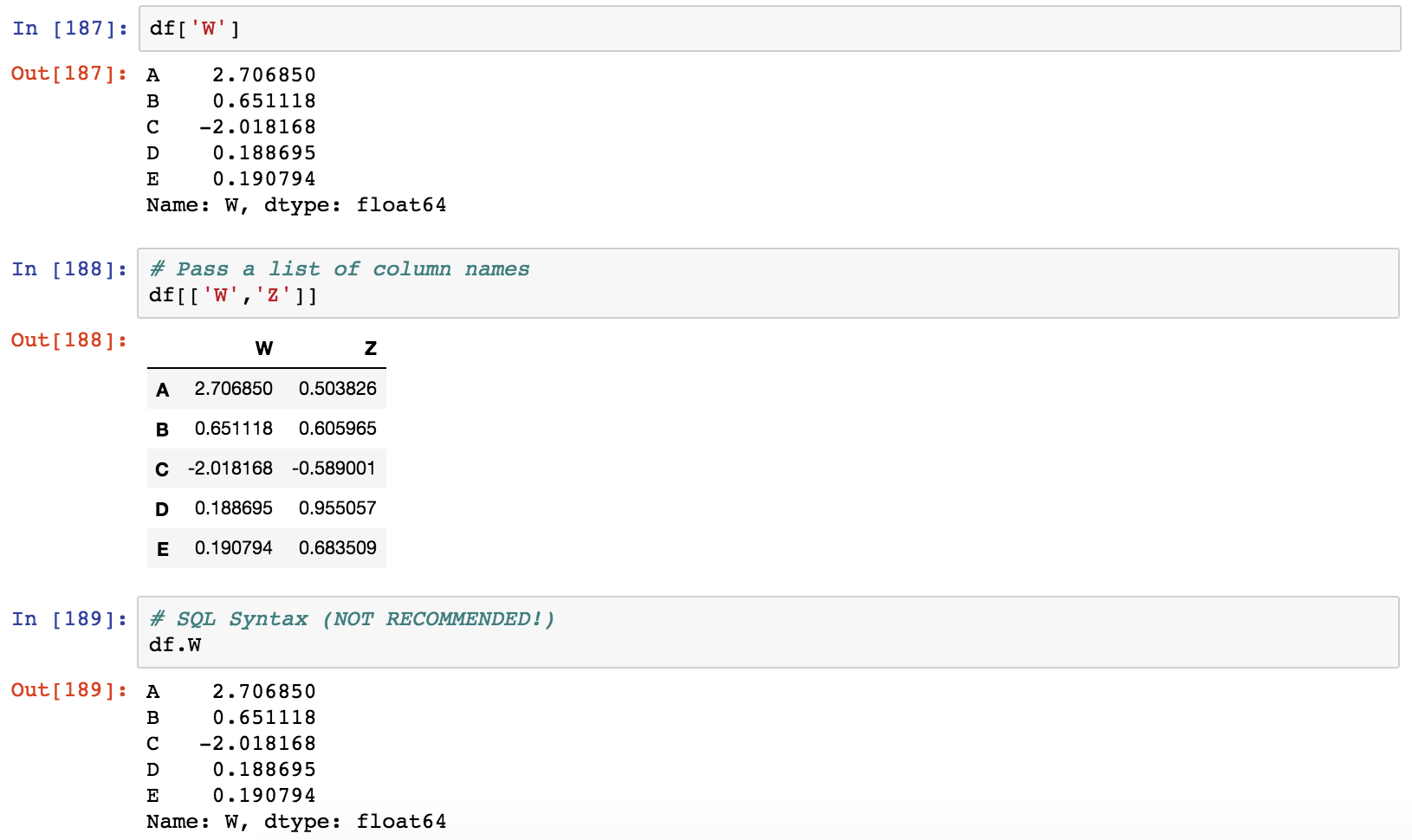
### DataFrames

DataFrames are the workhorse of pandas and are directly inspired by the R programming language. We can think of a DataFrame as a bunch of Series objects put together to share the same index. Let's use pandas to explore this topic!

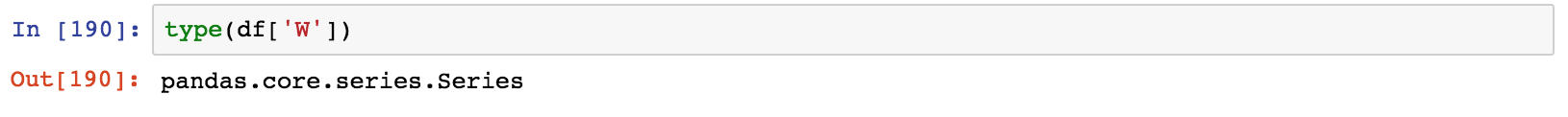


* 1. ***Selection and Indexing***

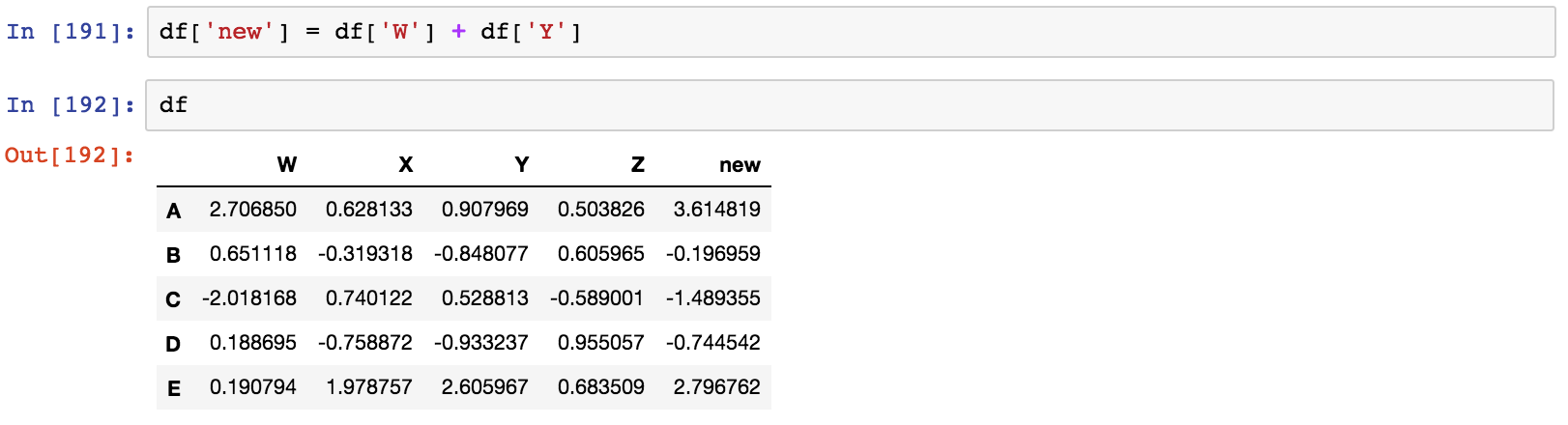
Let's learn the various methods to grab data from a DataFrame



DataFrame Columns are just Series

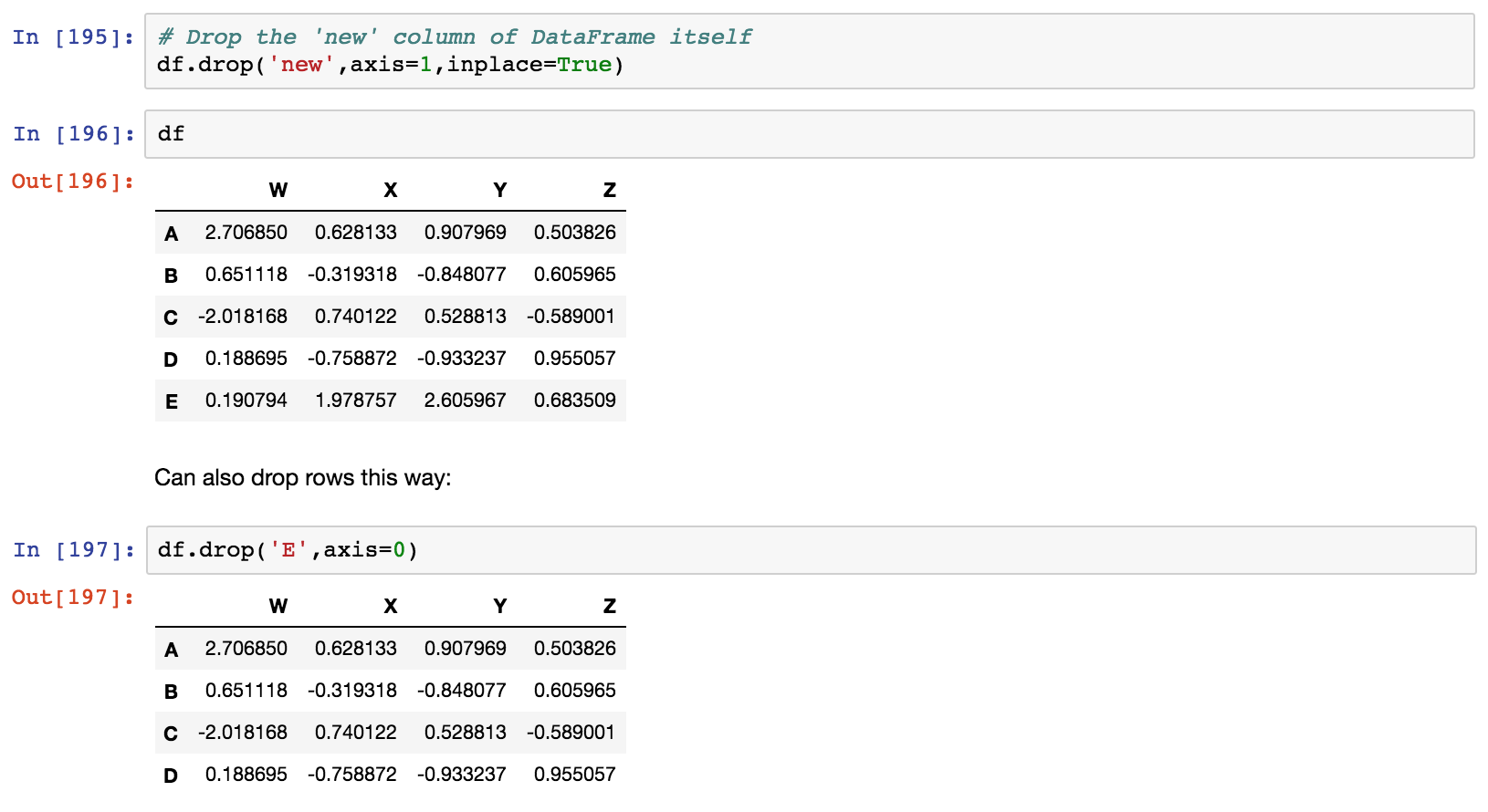


Creating a new column:

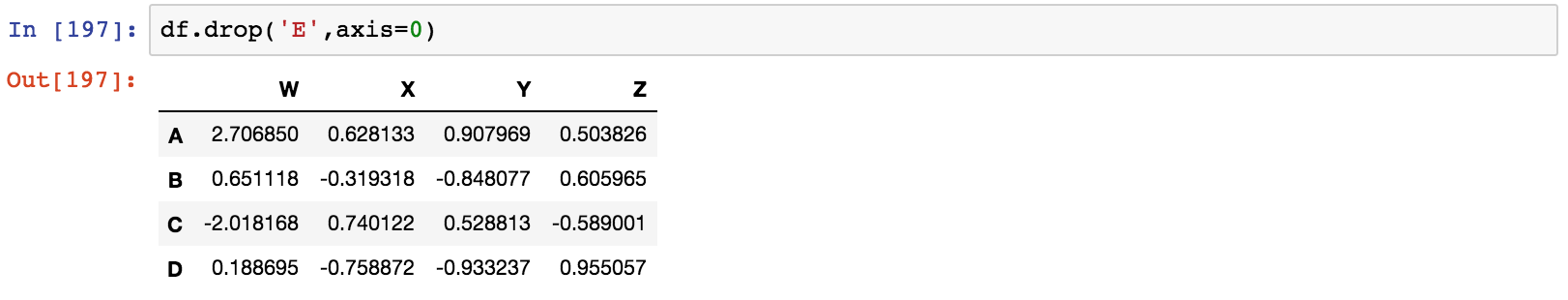


Removing Columns

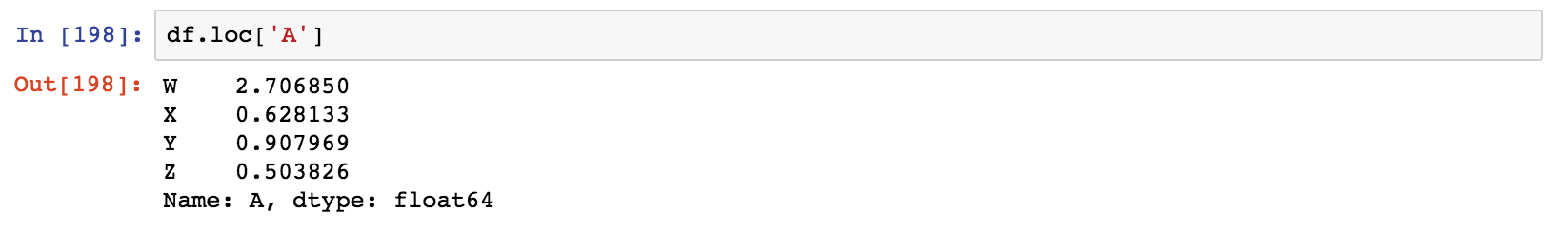




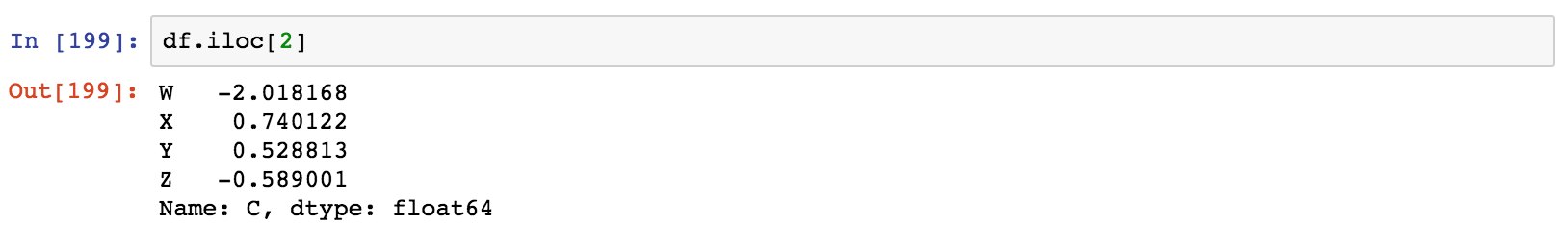
Can also drop rows this way:



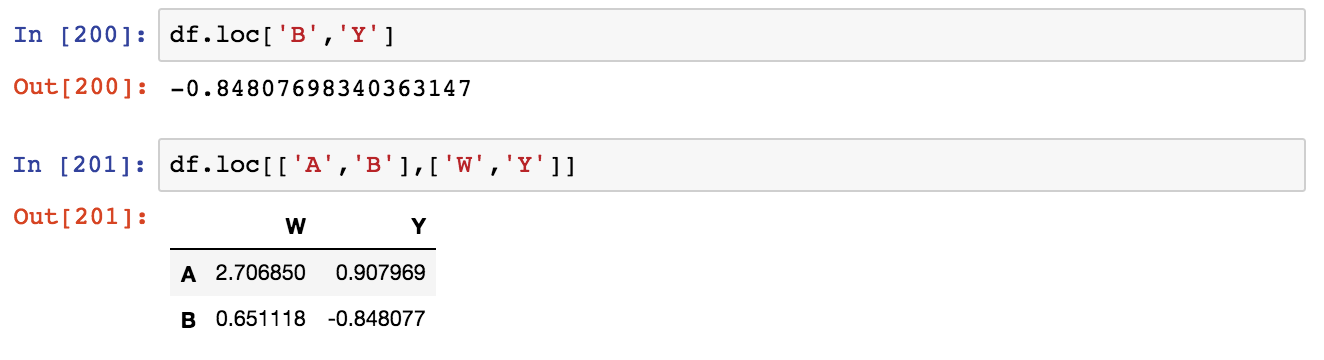
Selecting Rows



Or select based off of position instead of label

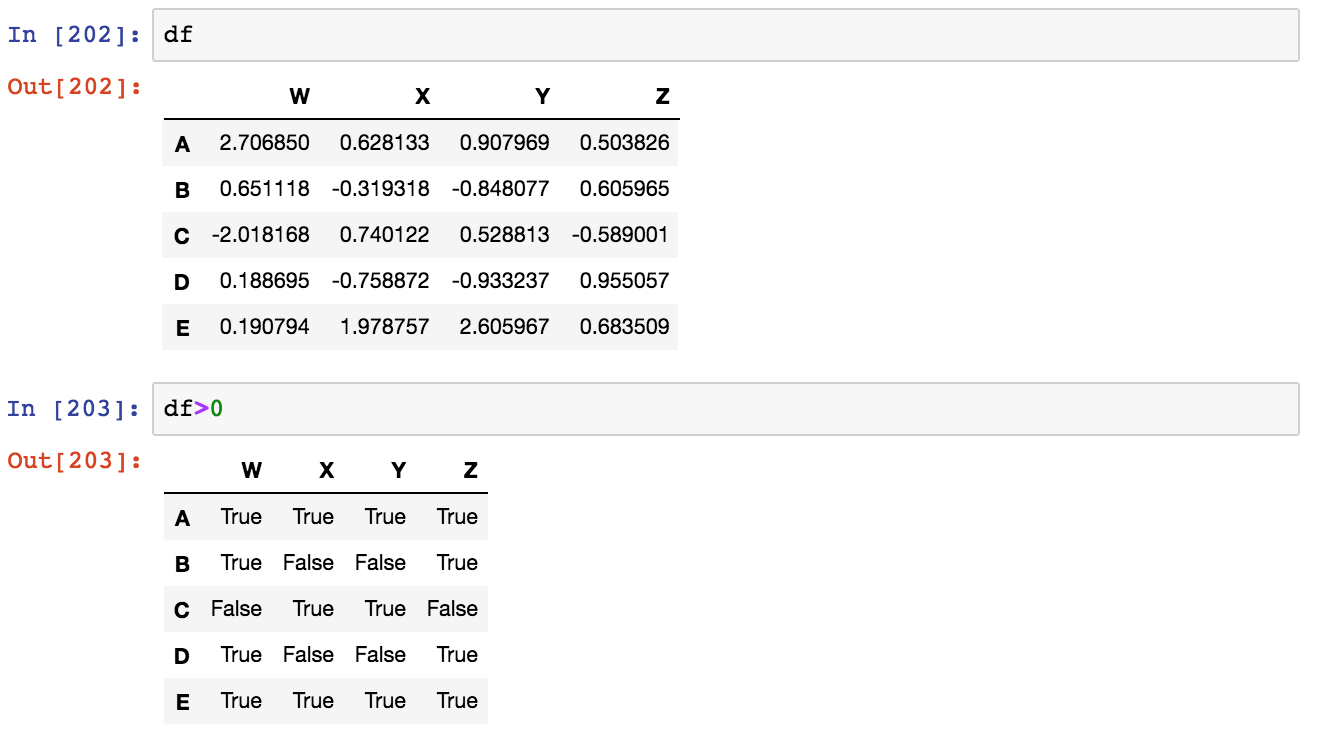


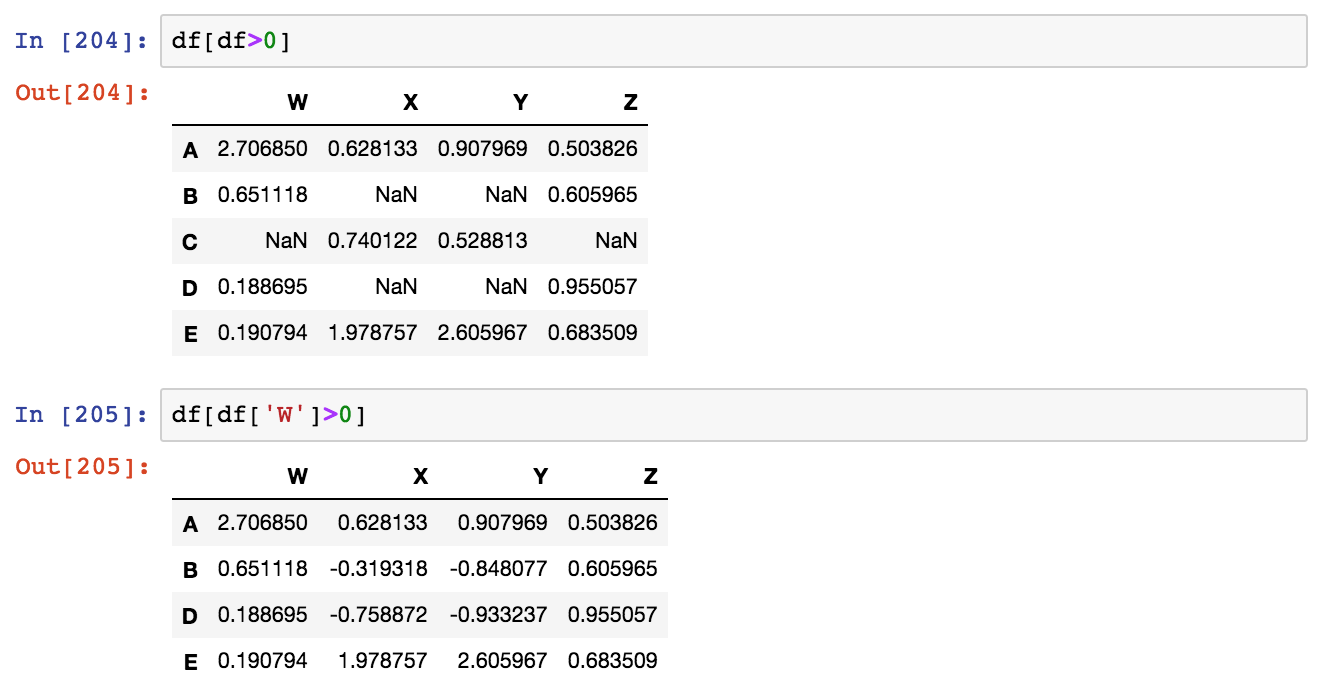
Selecting subset of rows and columns

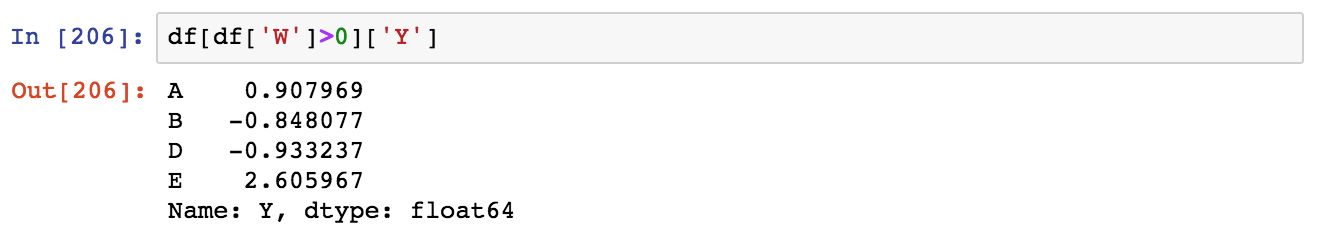


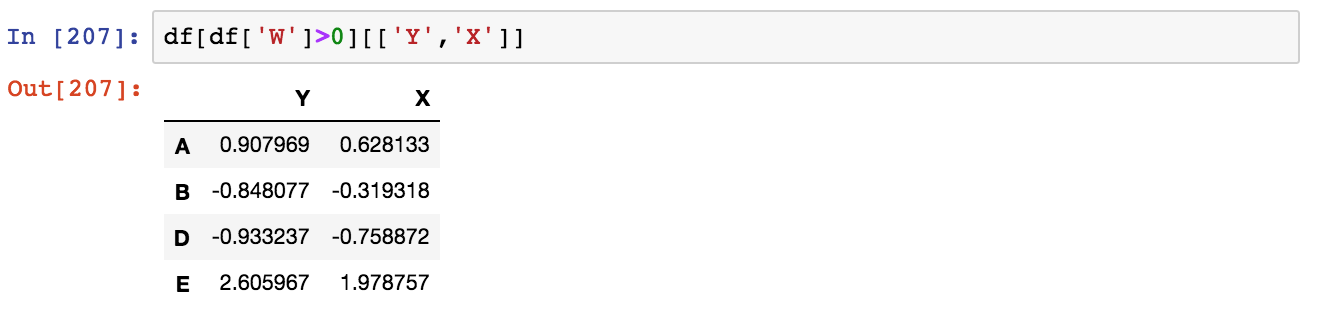
* 1. ***Conditional Selection***

An important feature of pandas is conditional selection using bracket notation, very similar to numpy:

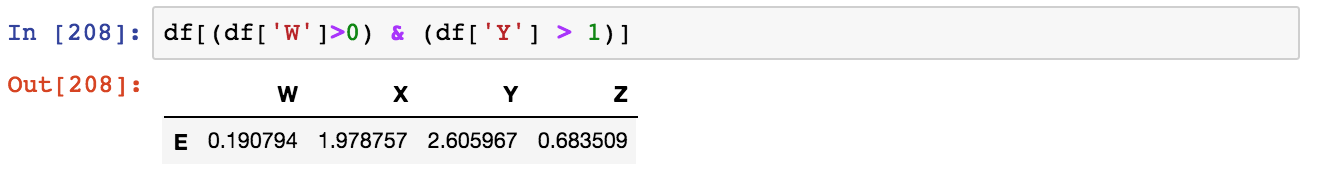






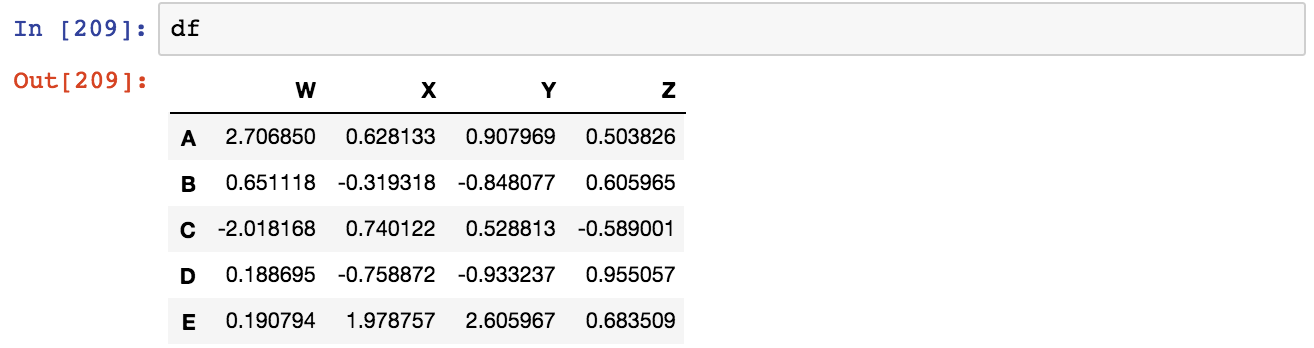


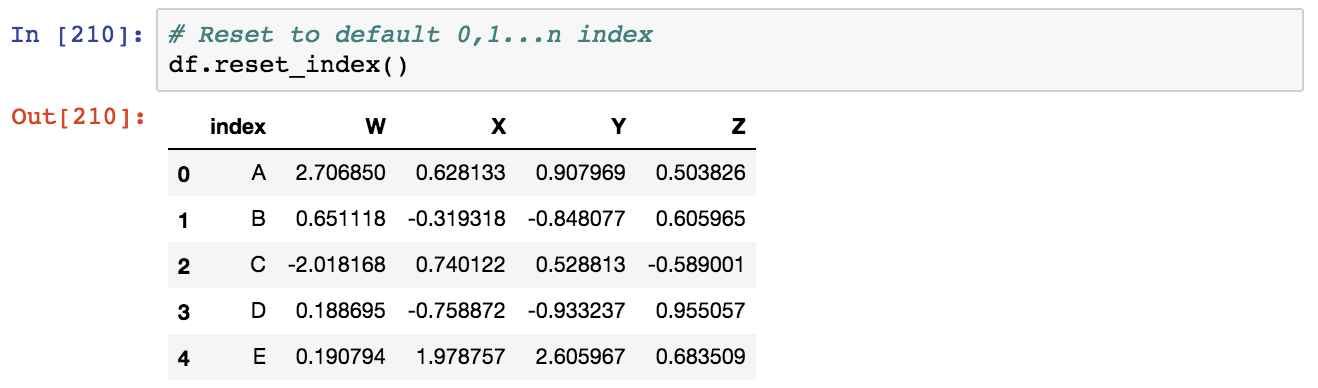
For two conditions you can use | and & with parenthesis:

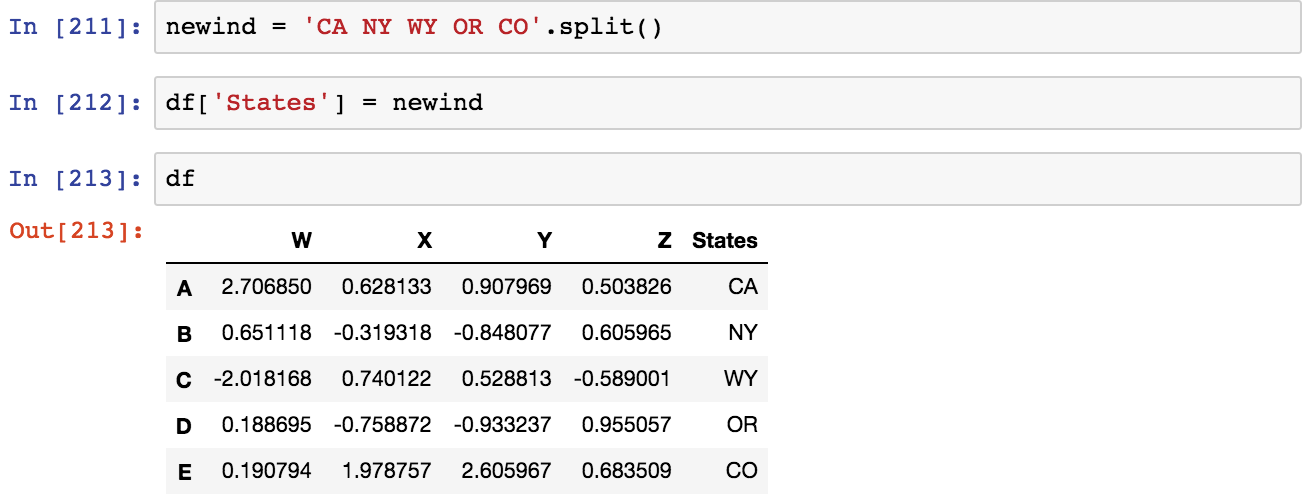


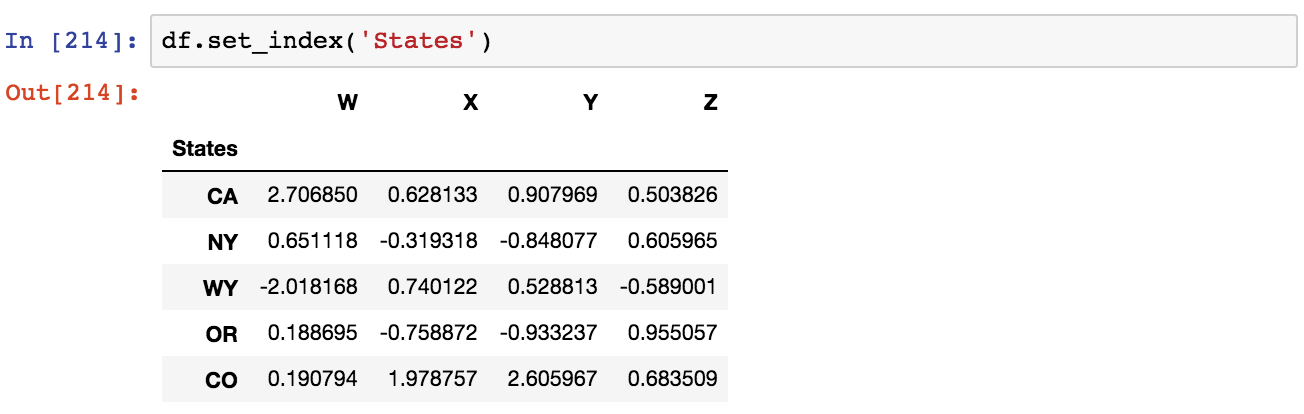
* 1. ***More Index Details***

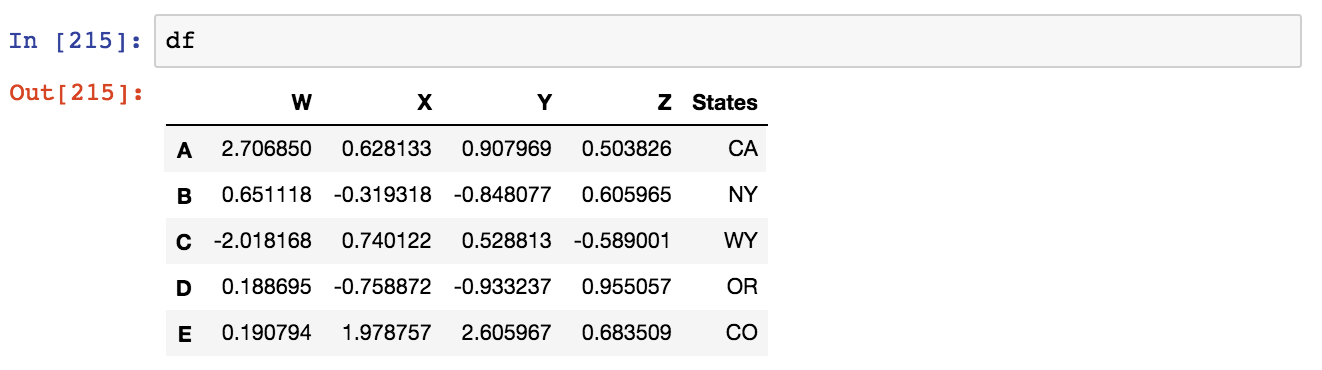
Let's discuss some more features of indexing, including resetting the index or setting it something else. We'll also talk about index hierarchy!

******

******

******

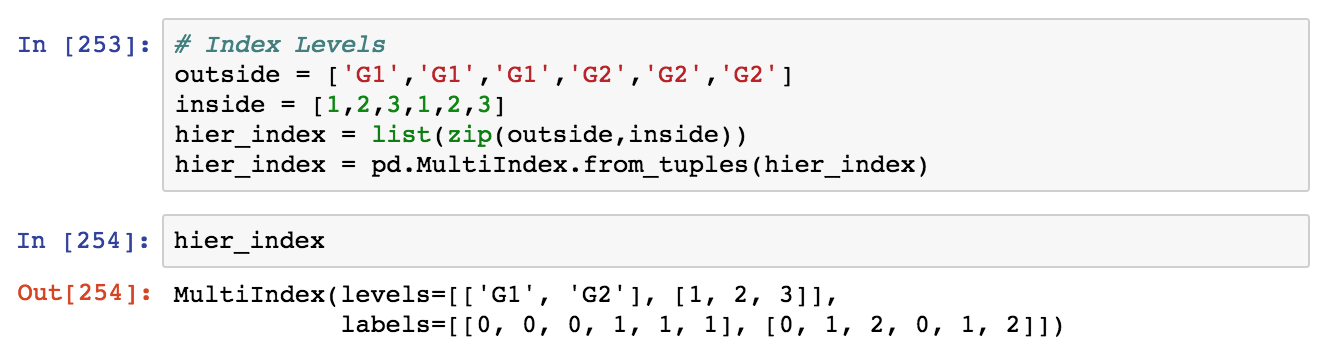
******

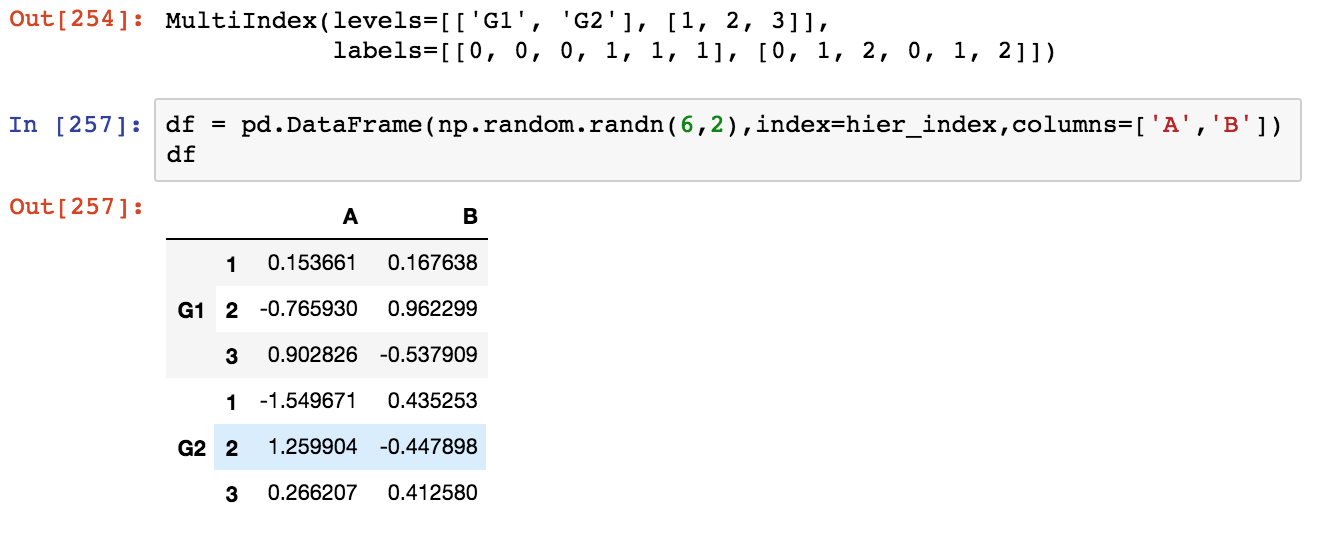
******

******

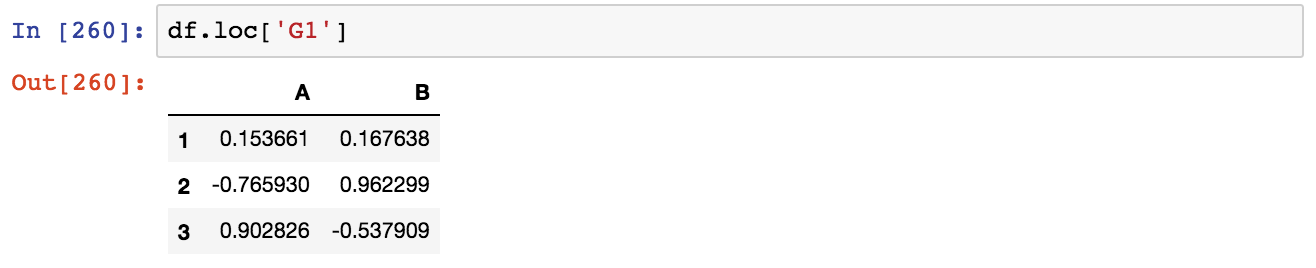
* 1. ***Multi-Index and Index Hierarchy***

Let us go over how to work with Multi-Index, first we'll create a quick example of what a Multi-Indexed DataFrame would look like:

******

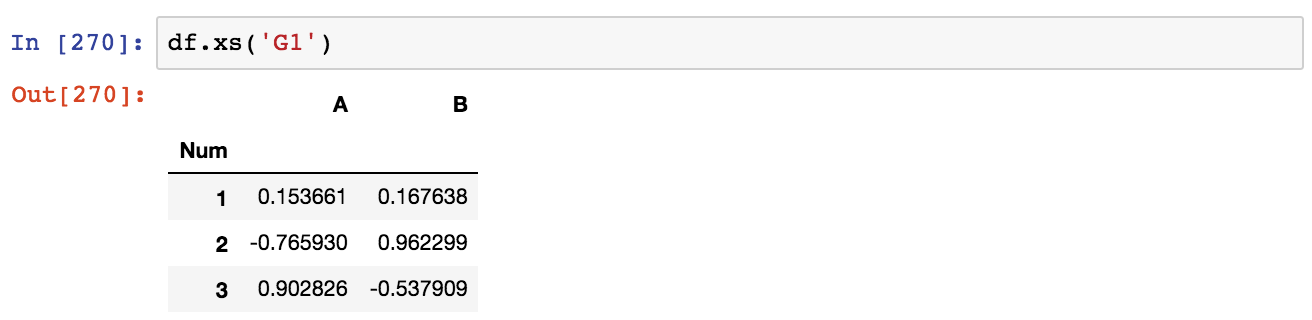
******

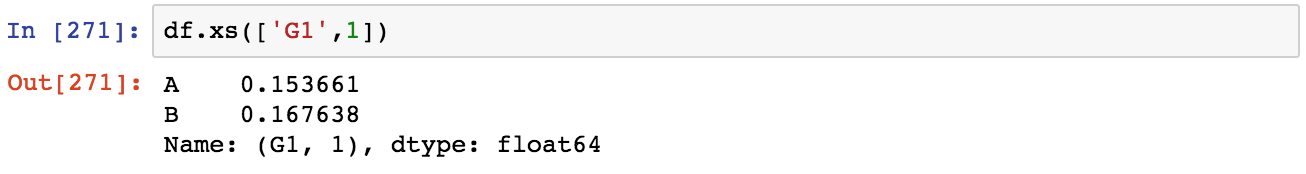
Now let's show how to index this! For index hierarchy we use df.loc[], if this was on the columns axis, you would just use normal bracket notation df[]. Calling one level of the index returns the sub-dataframe:

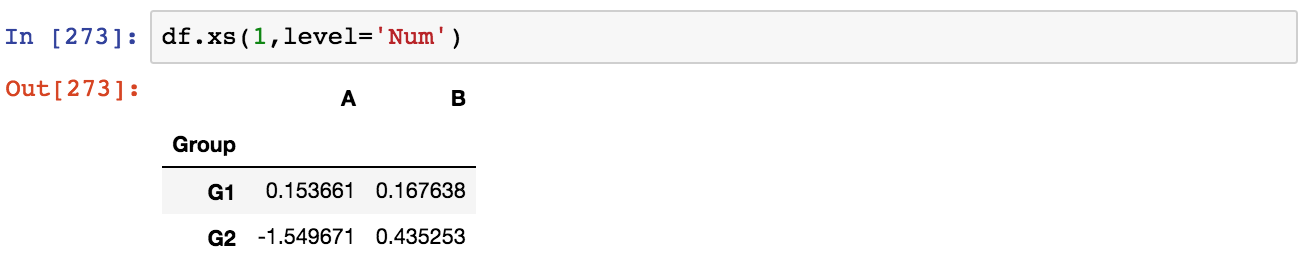
******

******

******

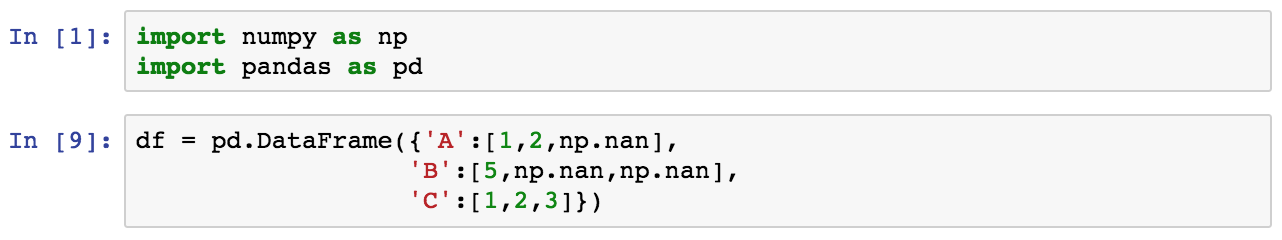
******

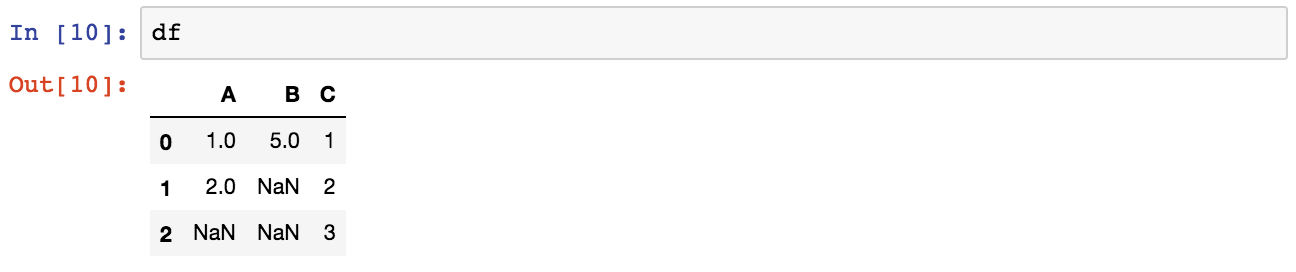
******

******

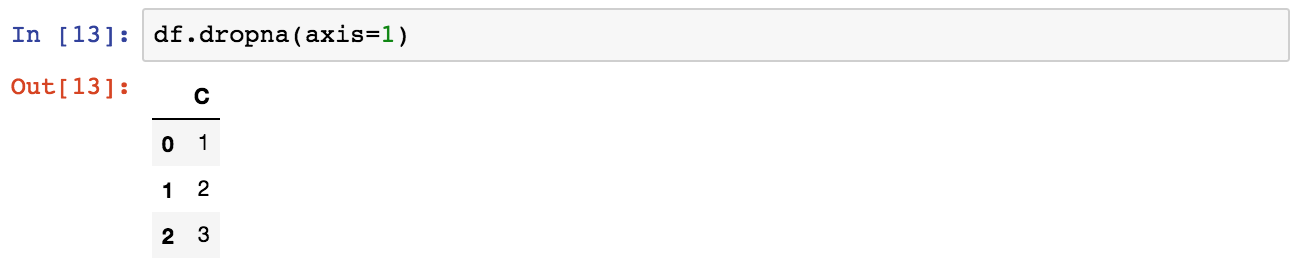
### Missing Data

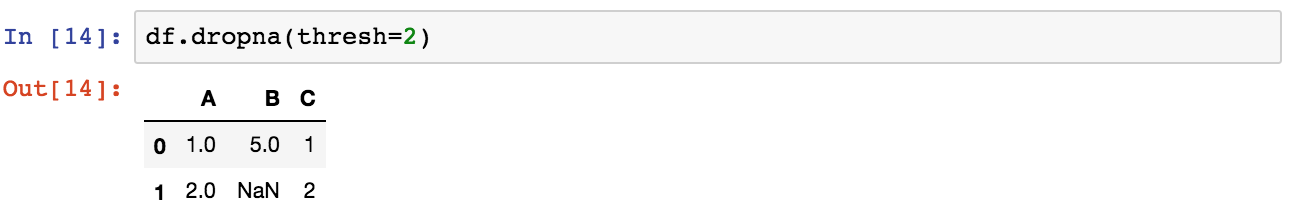
Let's show a few convenient methods to deal with Missing Data in pandas:

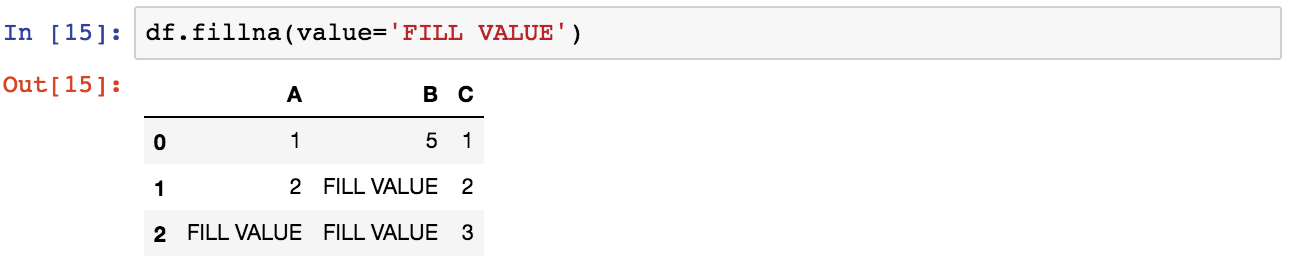


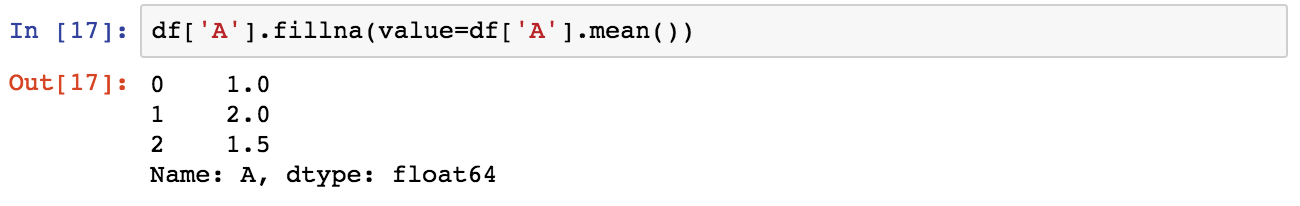








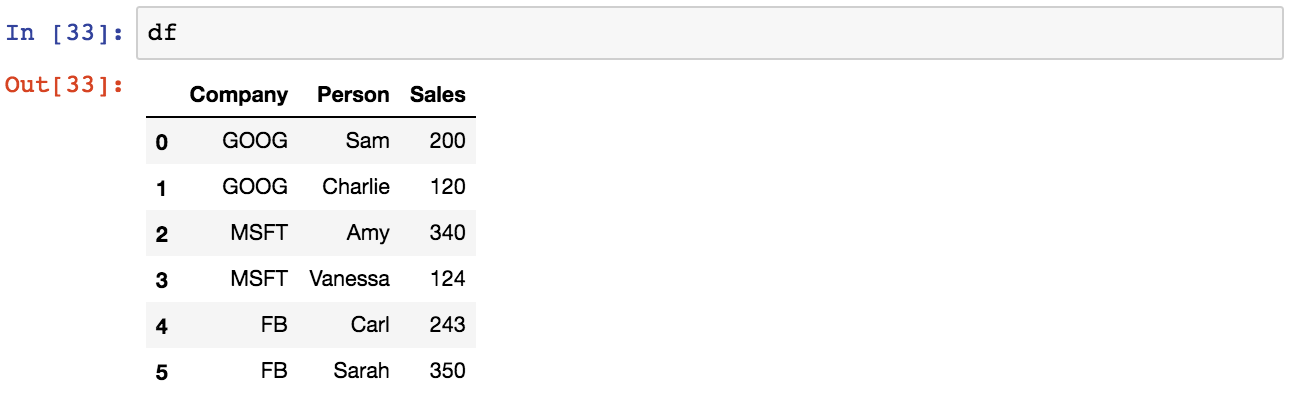




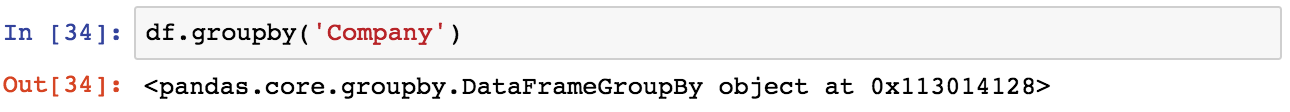
### GroupBy

The groupby method allows you to group rows of data together and call aggregate functions

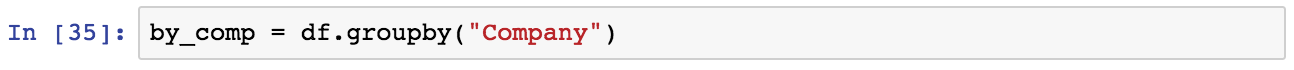




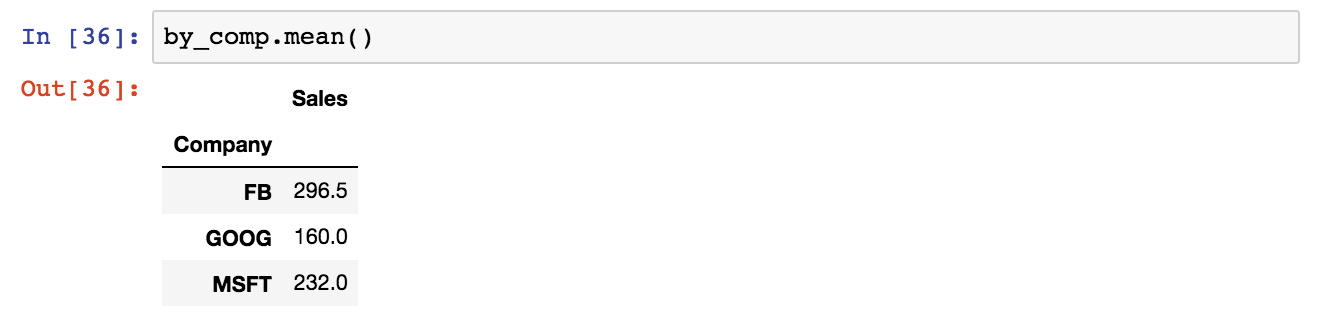
Now you can use the .groupby() method to group rows together based off of a column name. For instance let's group based off of Company. This will create a DataFrameGroupBy object:



You can save this object as a new variable:

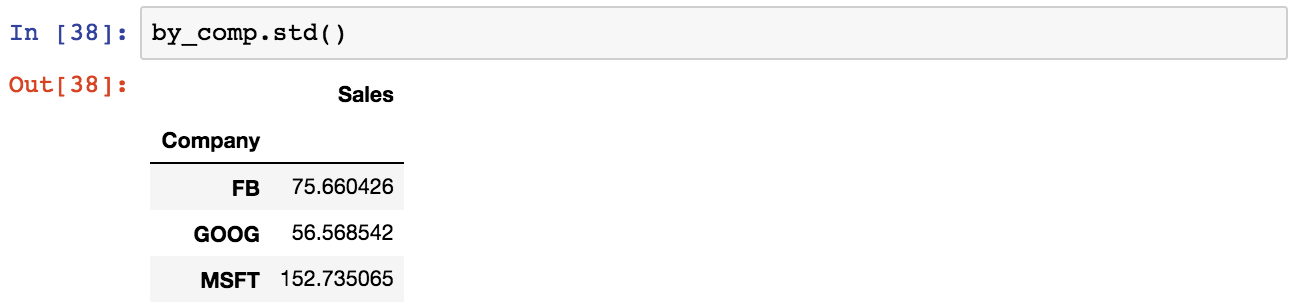


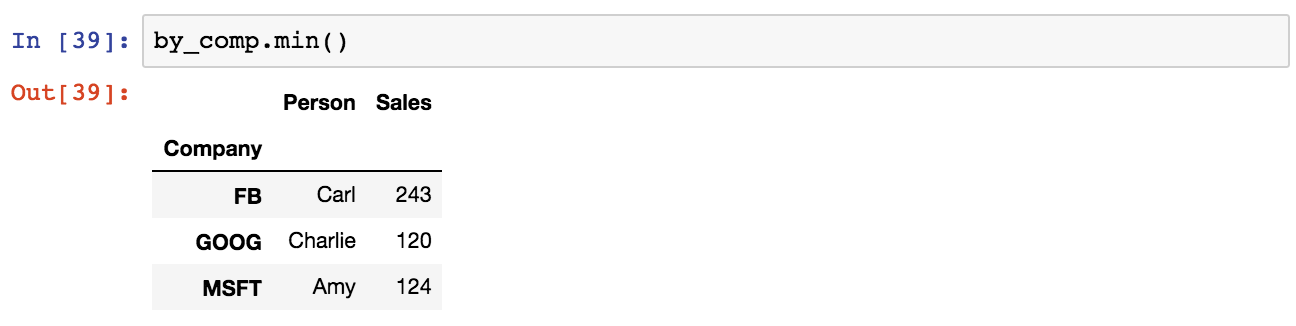
And then call aggregate methods off the object:

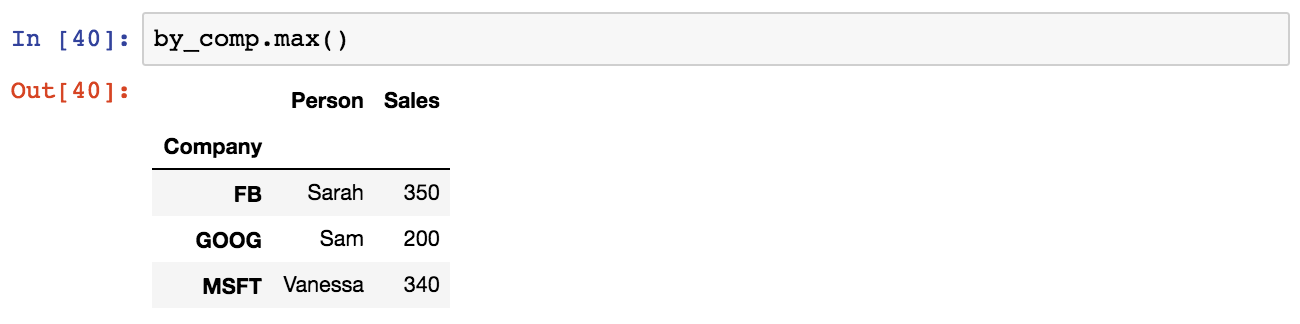


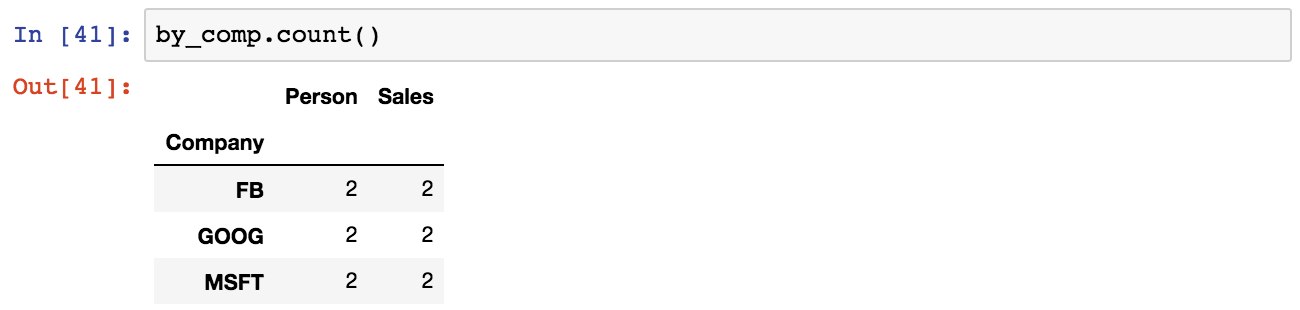


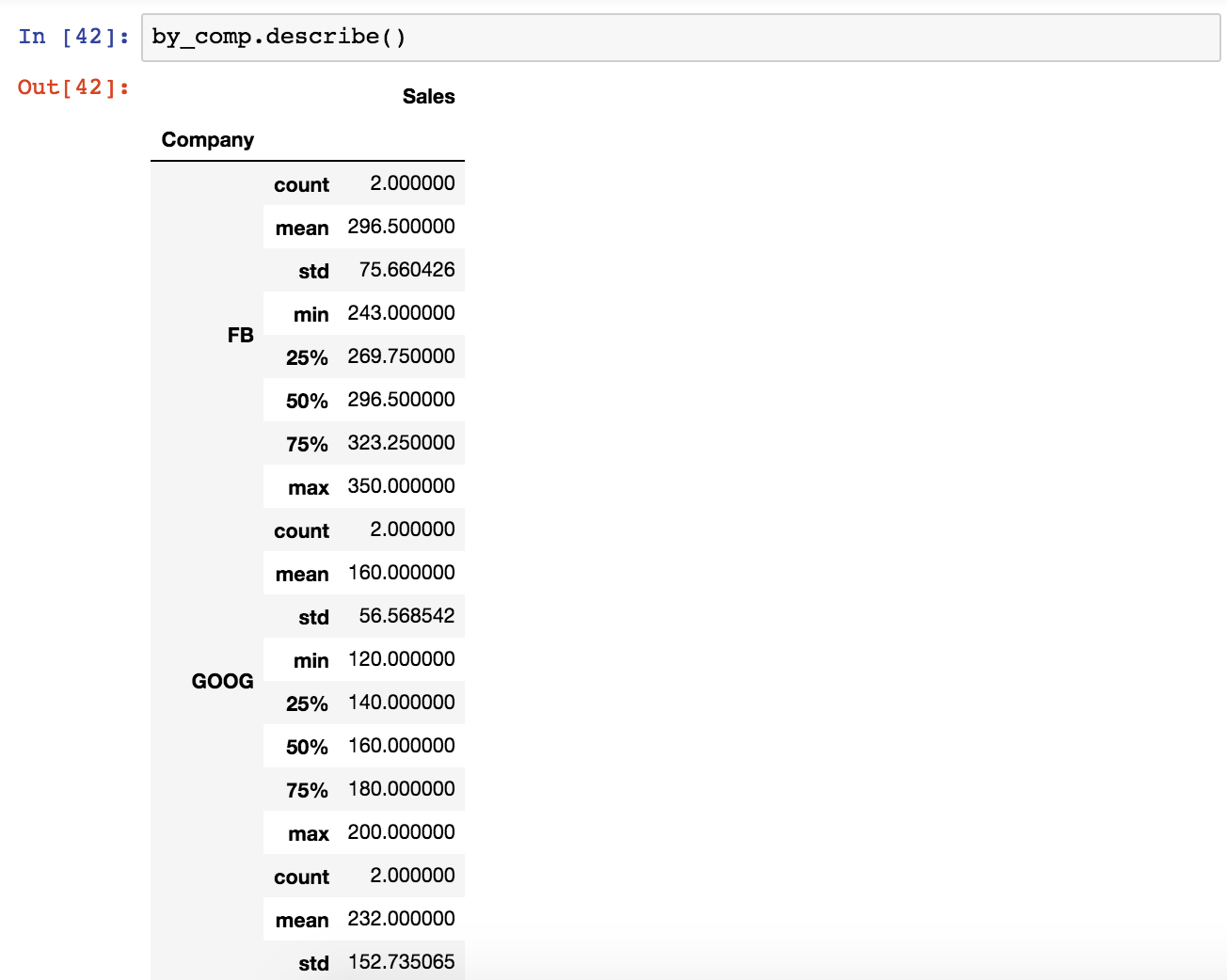
More examples of aggregate methods:

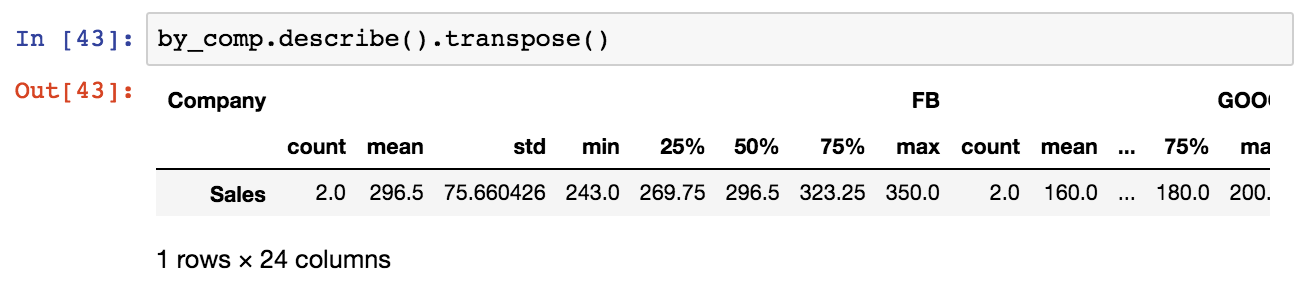


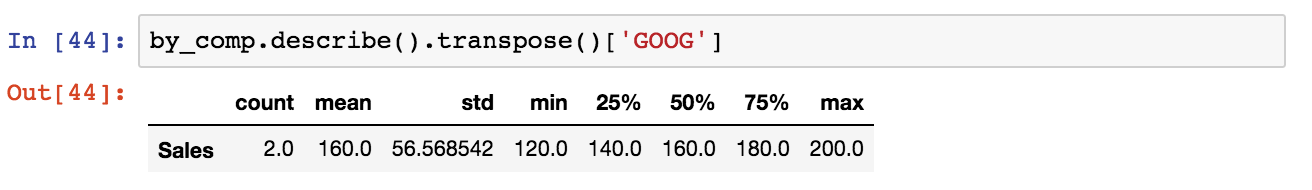










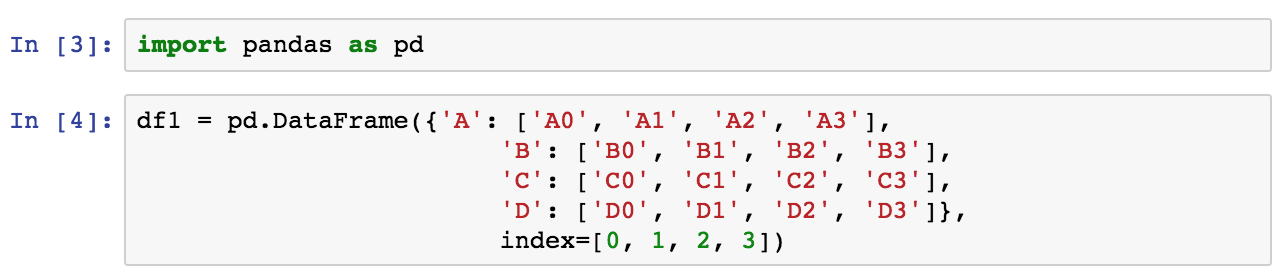


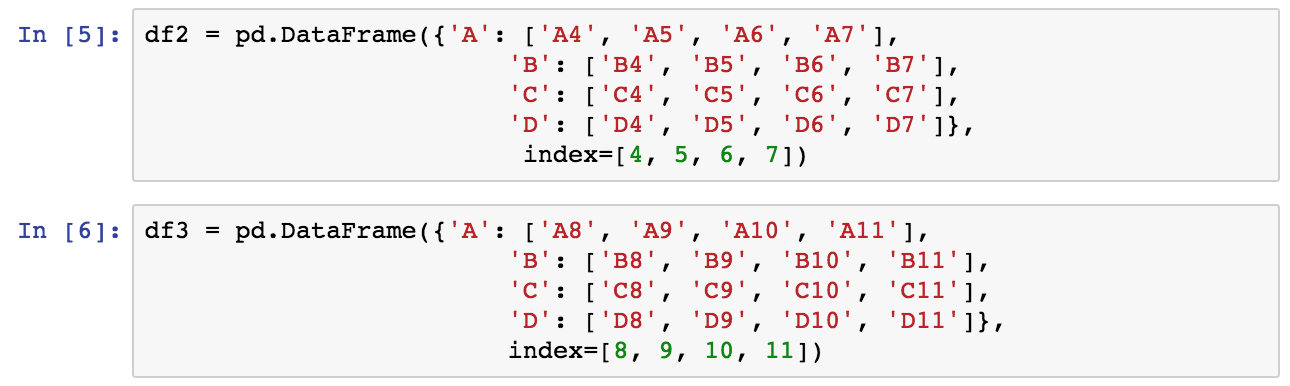
### Merging, Joining and Concatenating

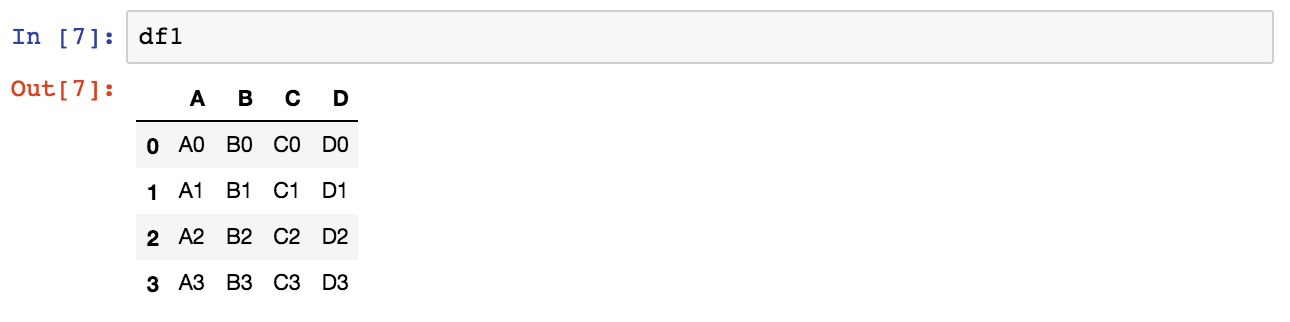
There are 3 main ways of combining DataFrames together: Merging, Joining and Concatenating. In this lecture we will discuss these 3 methods with examples.

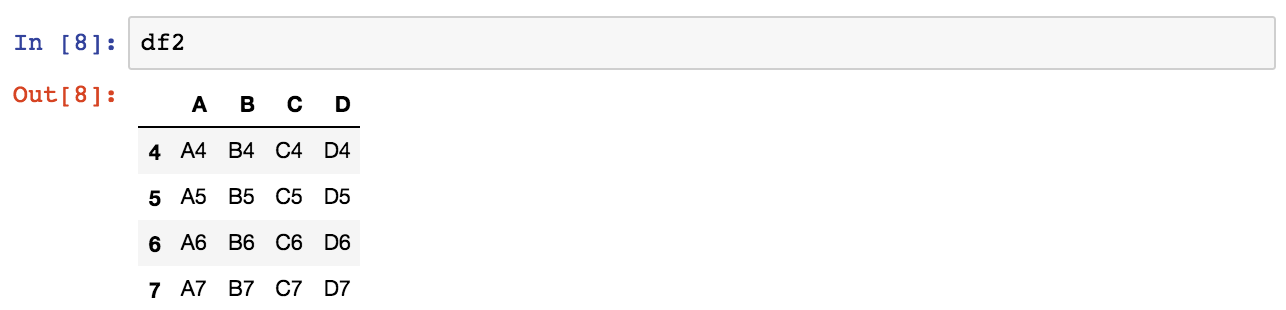
* 1. ***Concatenation***

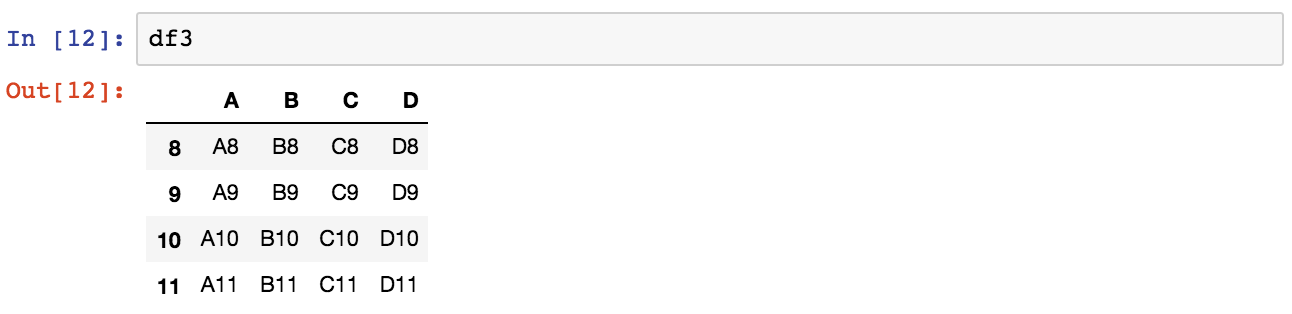
Example DataFrame

******

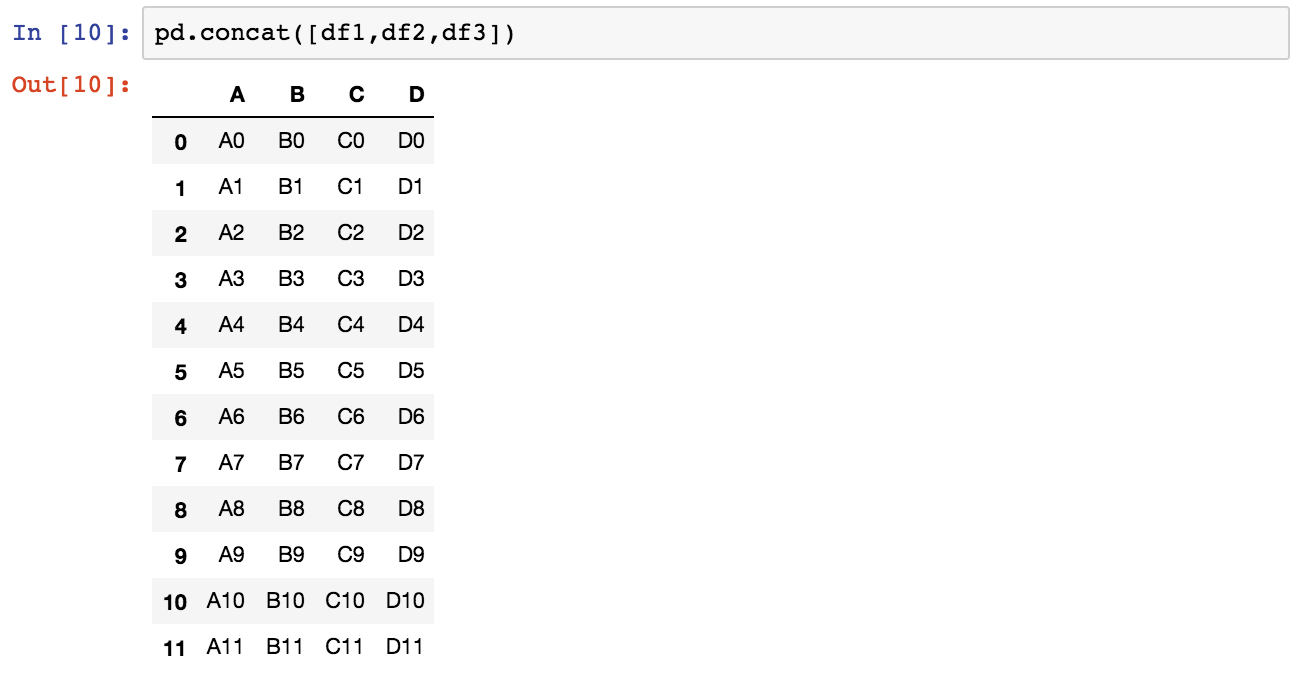
******

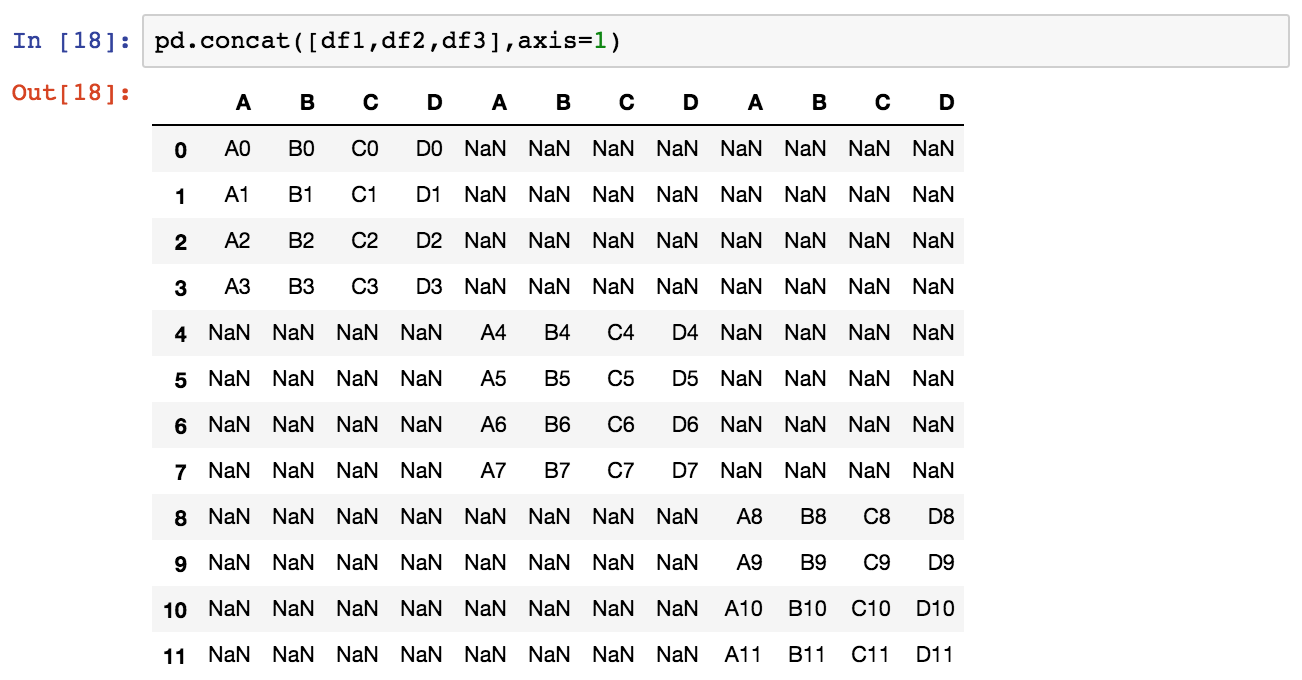
******

******

******

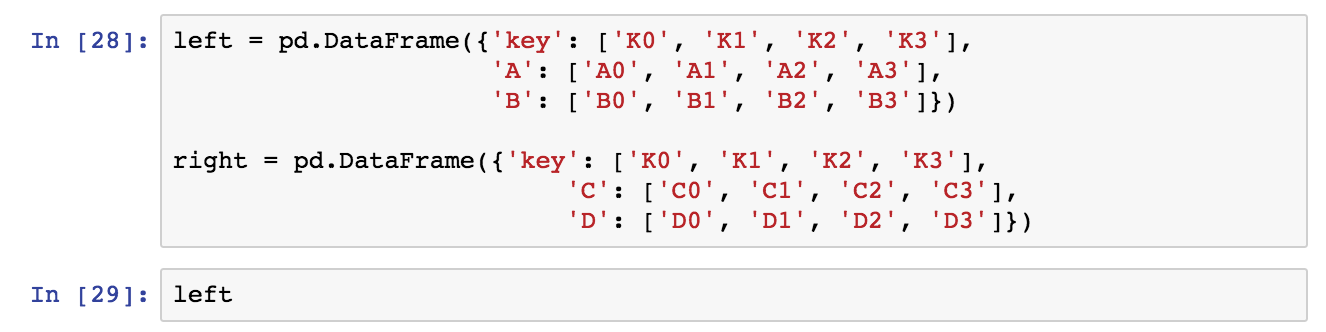
Concatenation basically glues together DataFrames. Keep in mind that dimensions should match along the axis you are concatenating on. You can use **pd.concat** and pass in a list of DataFrames to concatenate together:





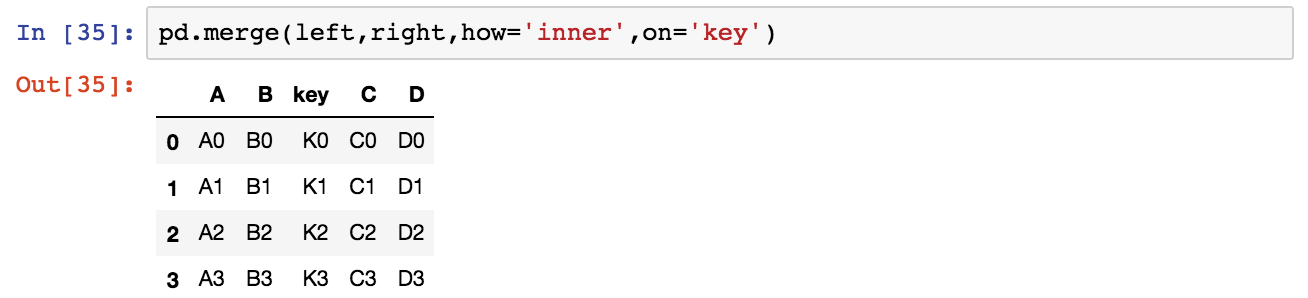
* 1. ***Merging***

Example DataFrame

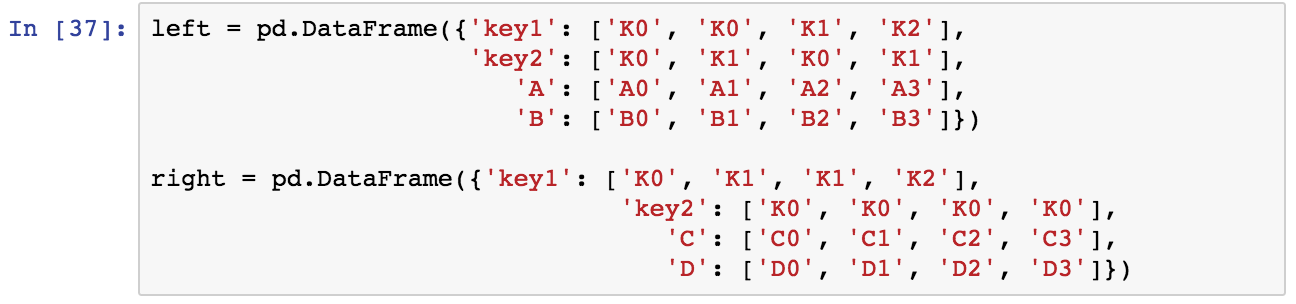
******

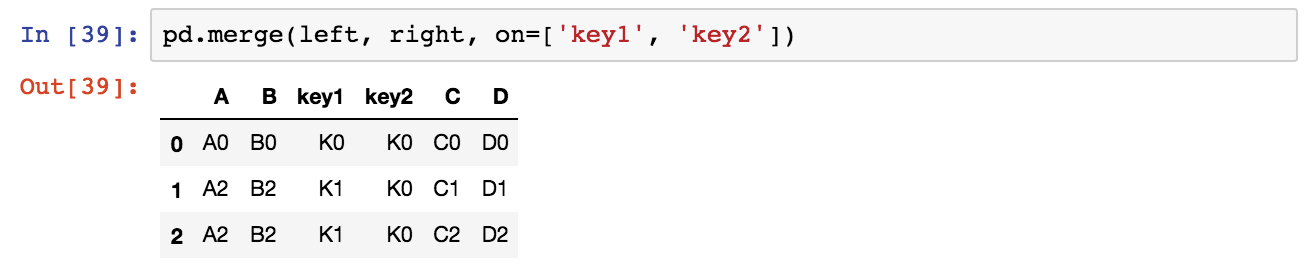
******

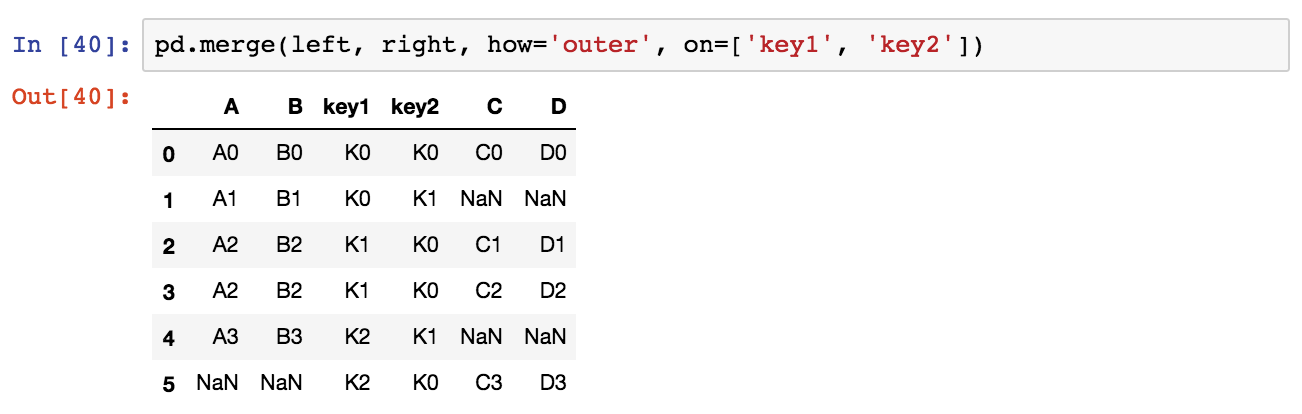
The **merge** function allows you to merge DataFrames together using a similar logic as merging SQL Tables together. For example:

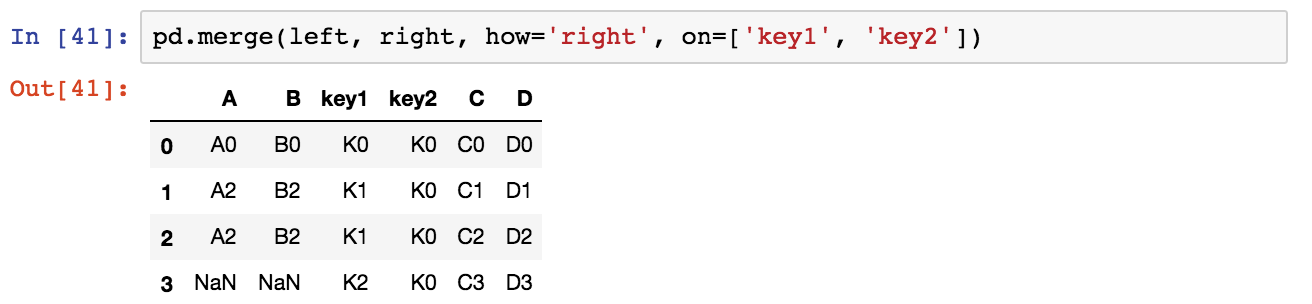
******

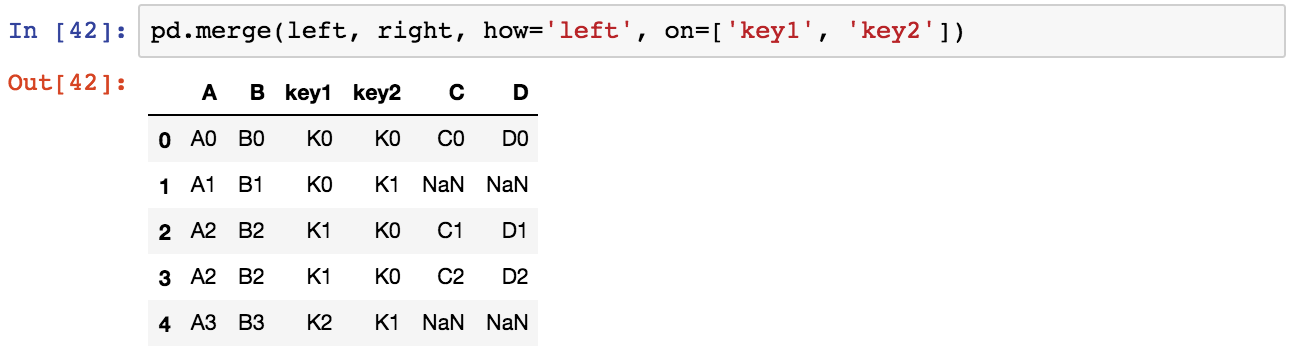
Or to show a more complicated example:





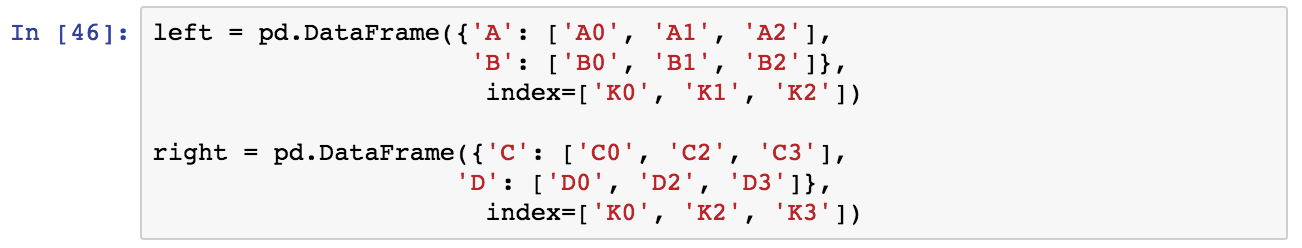


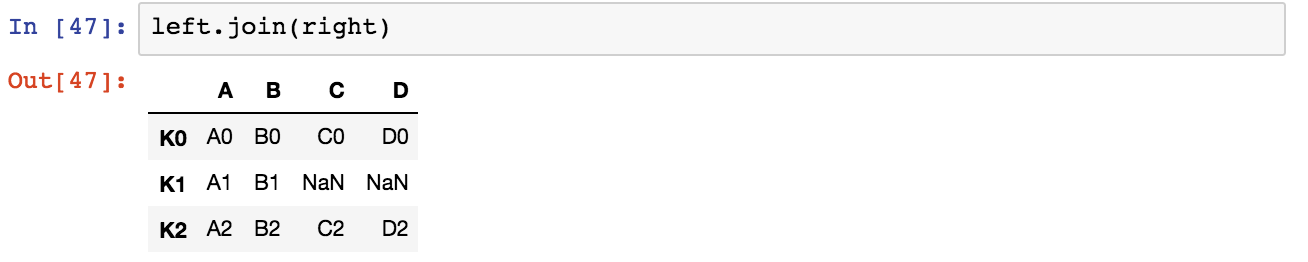


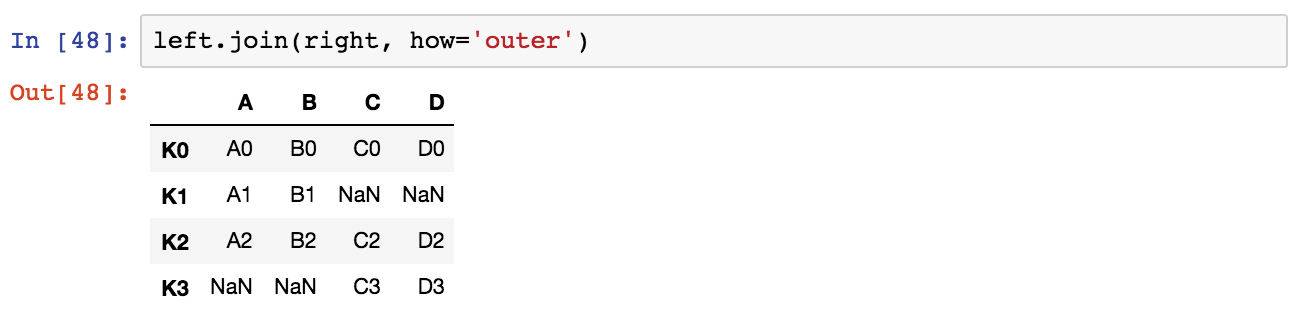


* 1. ***Joining***

Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

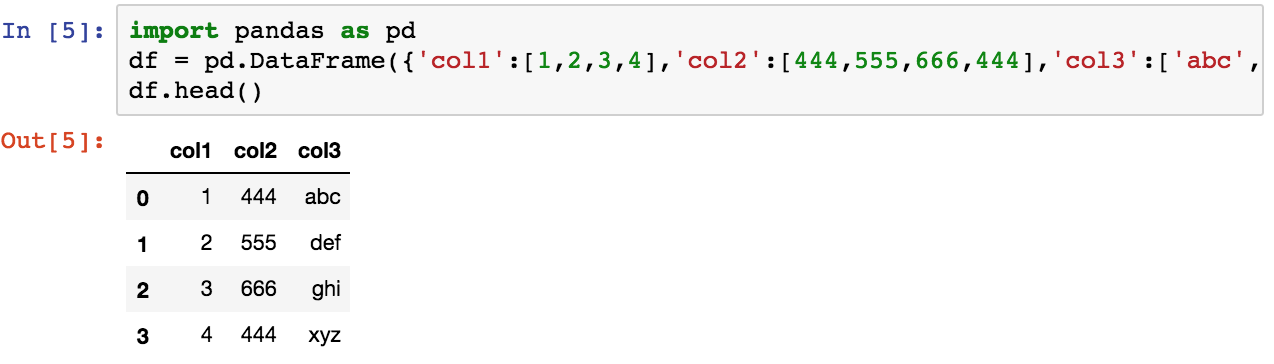
******

******

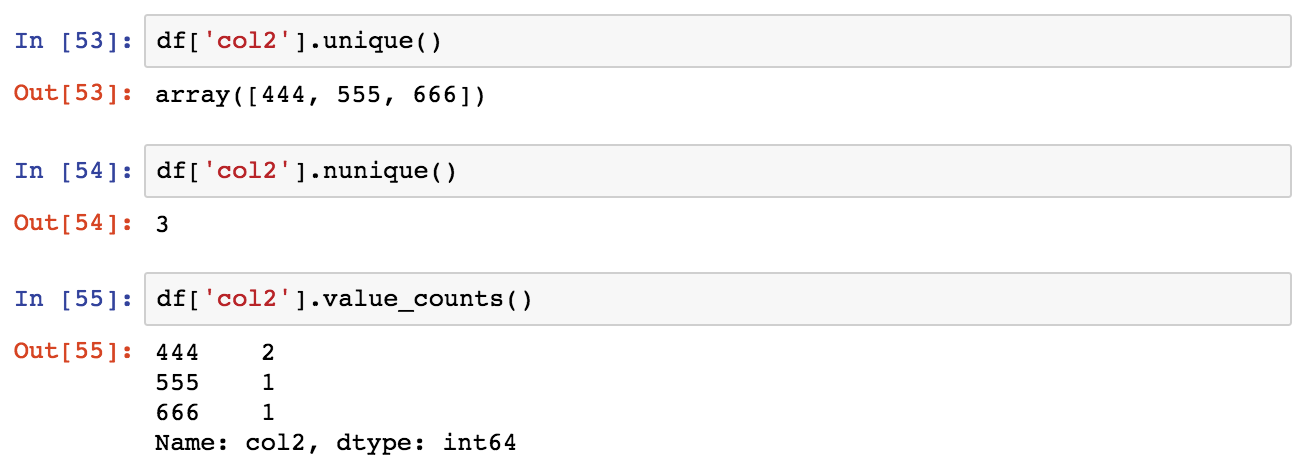
******

### Operations

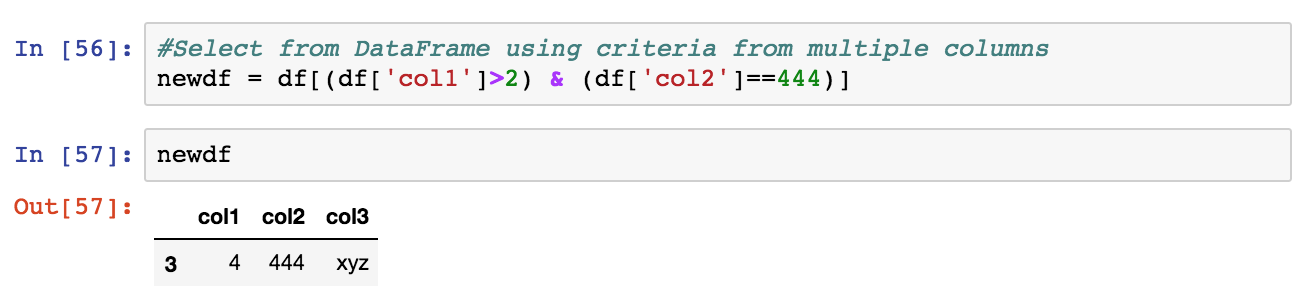
There are lots of operations with pandas that will be really useful to you, but don't fall into any distinct category. Let's show them here in this lecture:



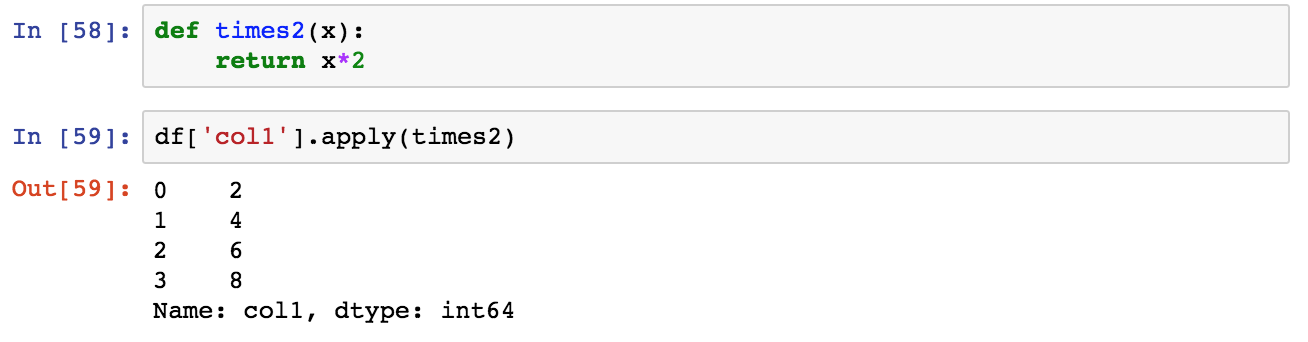
* 1. ***Info on Unique Values***

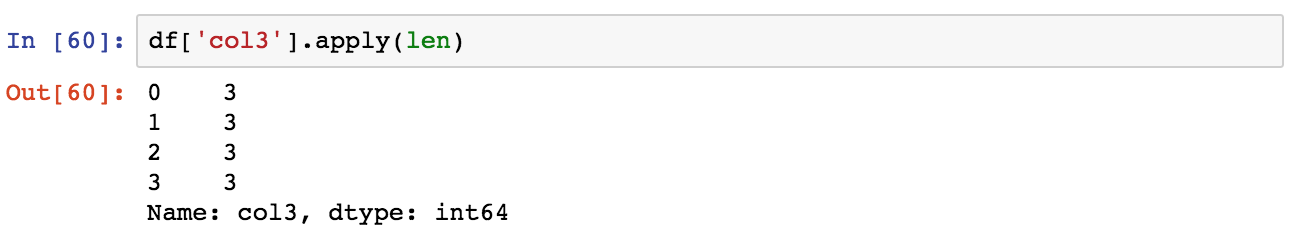
******

* 1. ***Selecting Data***

******

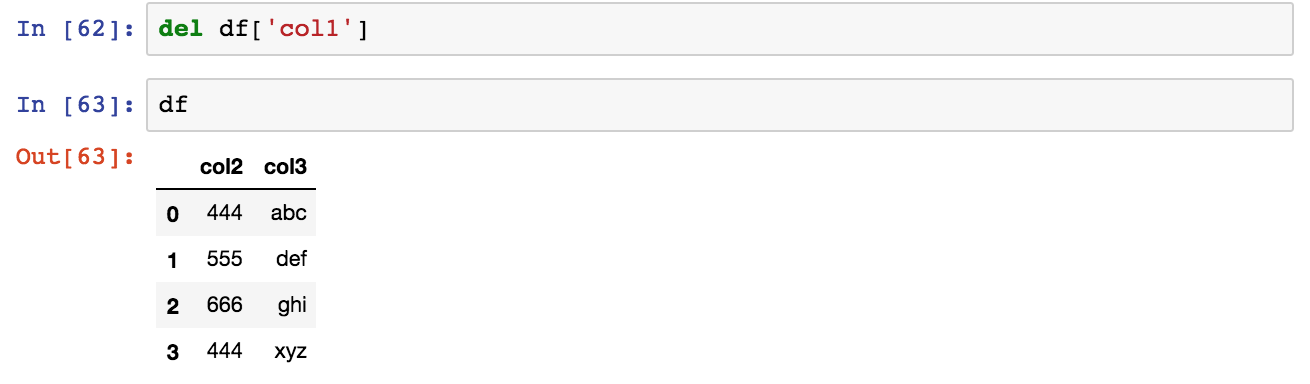
* 1. ***Applying Functions***

******

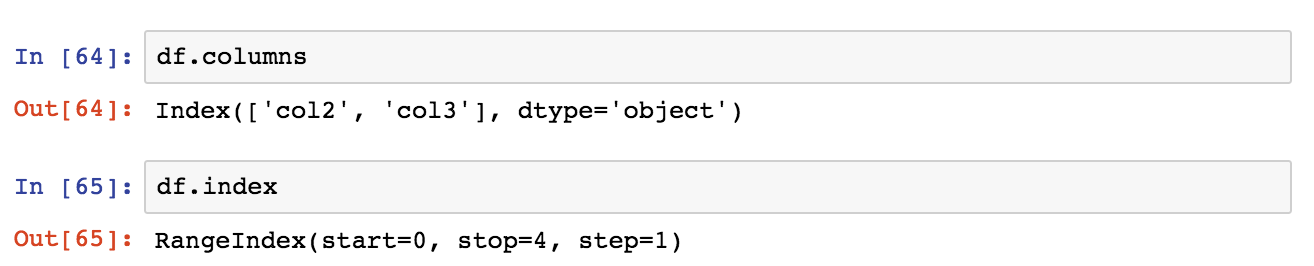
******

******

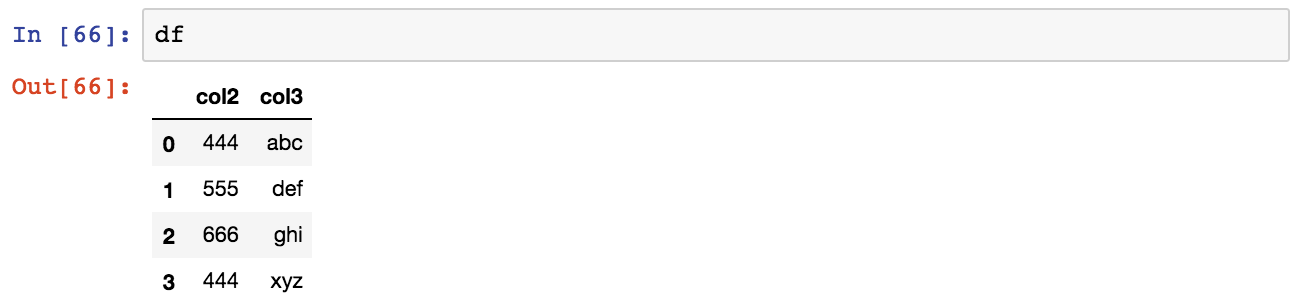
Permanently Removing a Column

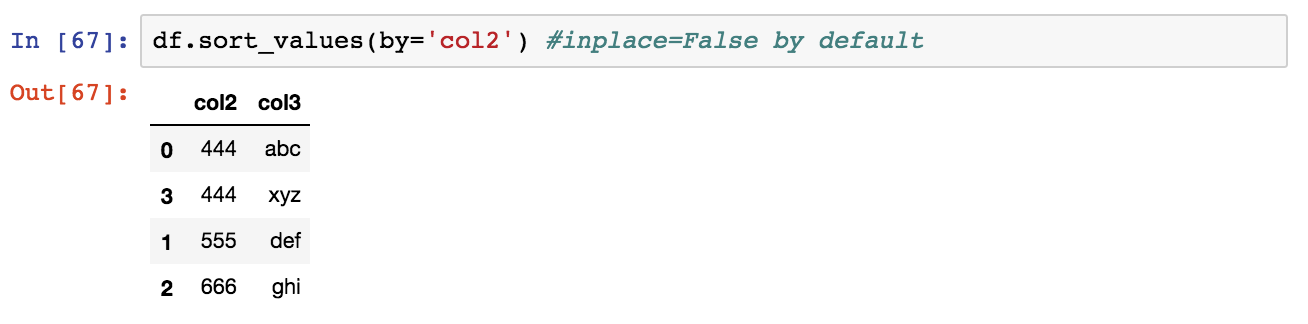
******

Get column and index names:

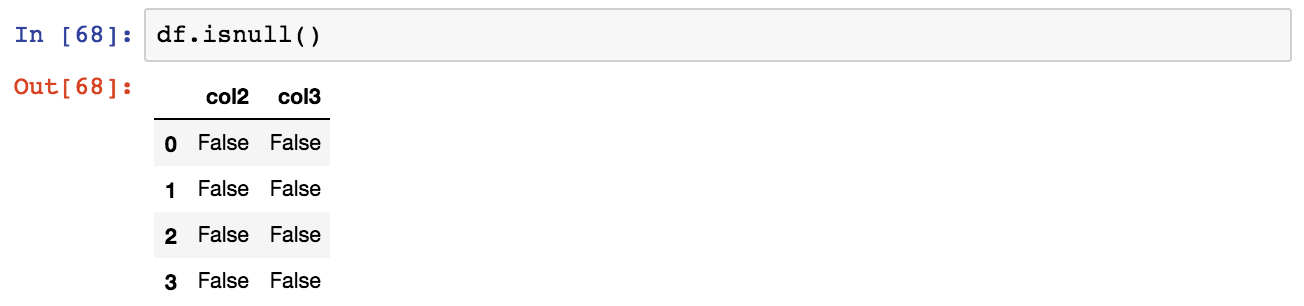
******

Sorting and Ordering a DataFrame:





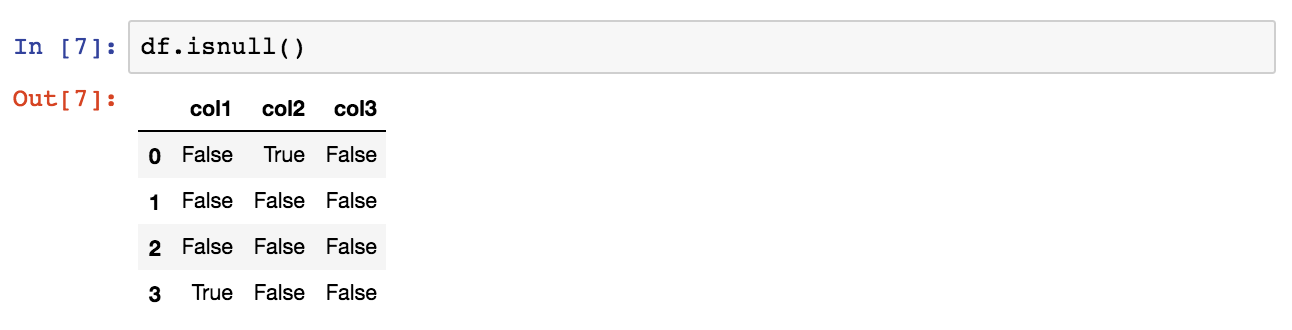
Find Null Values or Check for Null Values





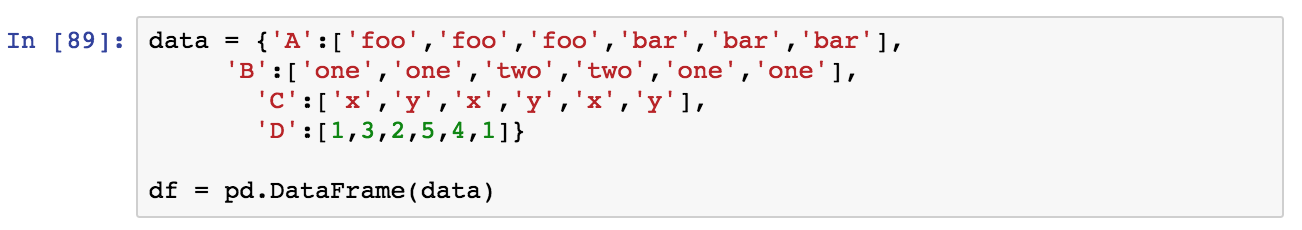
Filling in NaN values with something else:

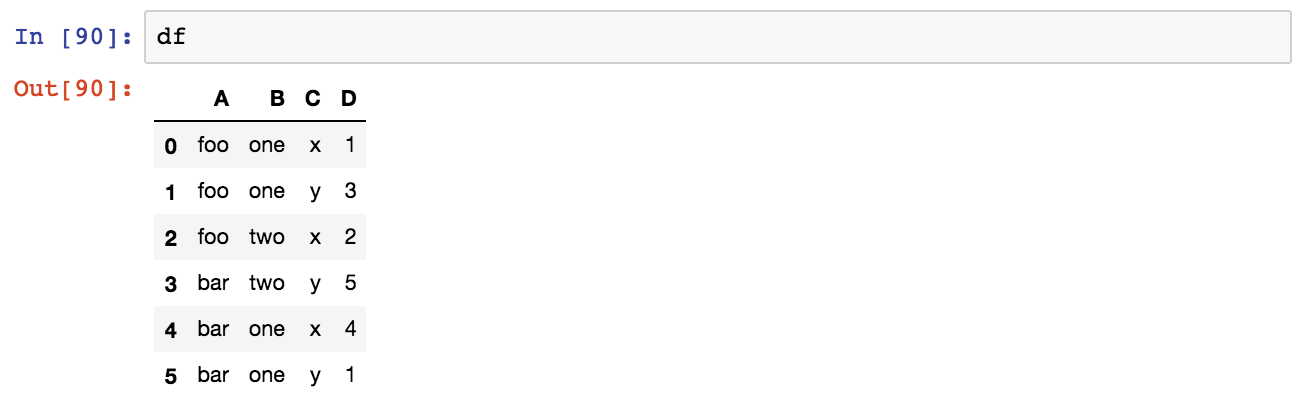


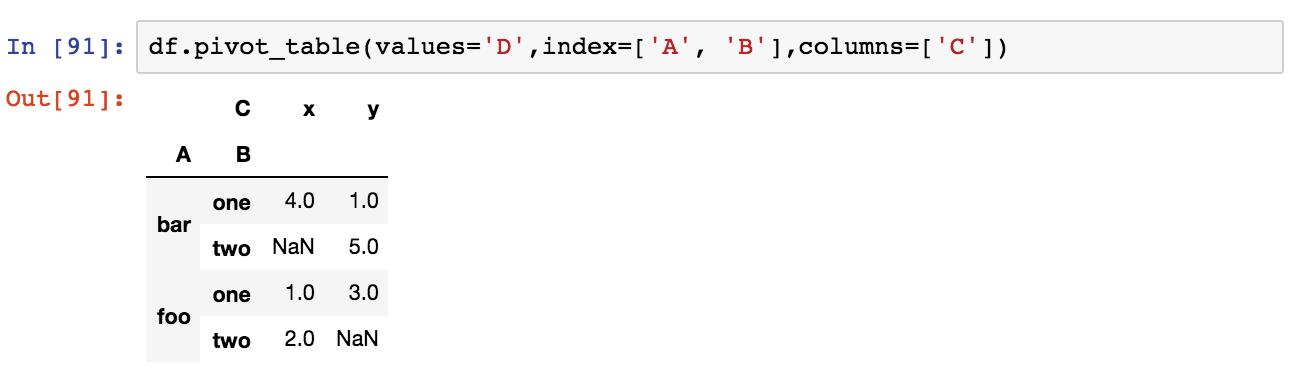






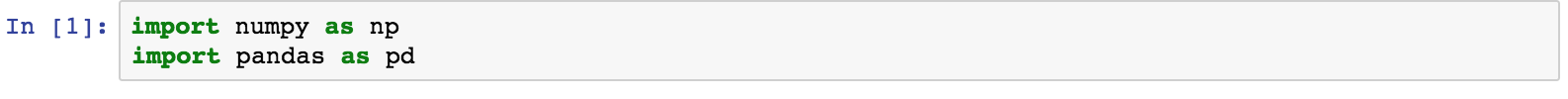






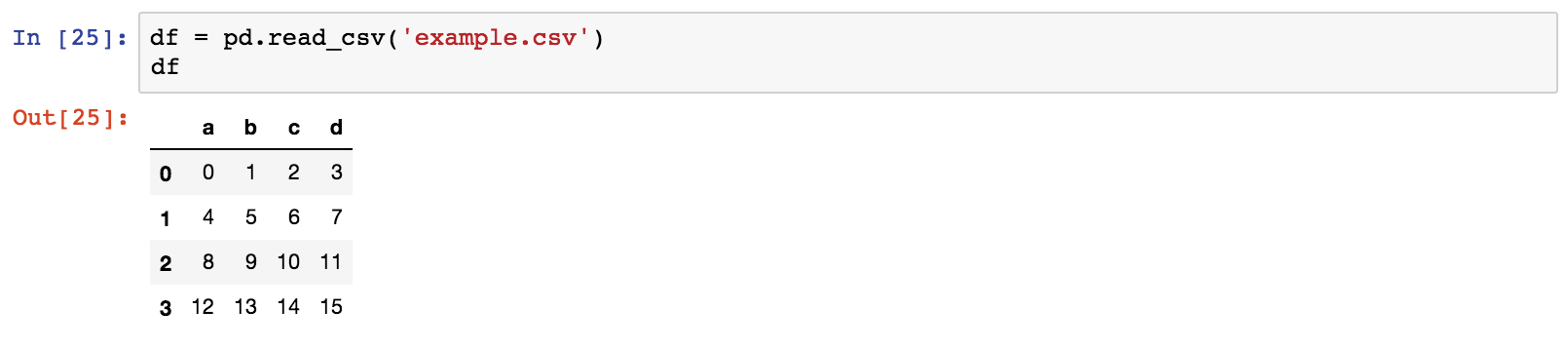
### Data Input and Output

This notebook is the reference code for getting input and output, pandas can read a variety of file types using its pd.read\_ methods. Let's take a look at the most common data types:

******

* 1. ***CSV***

**CSV Input**

******

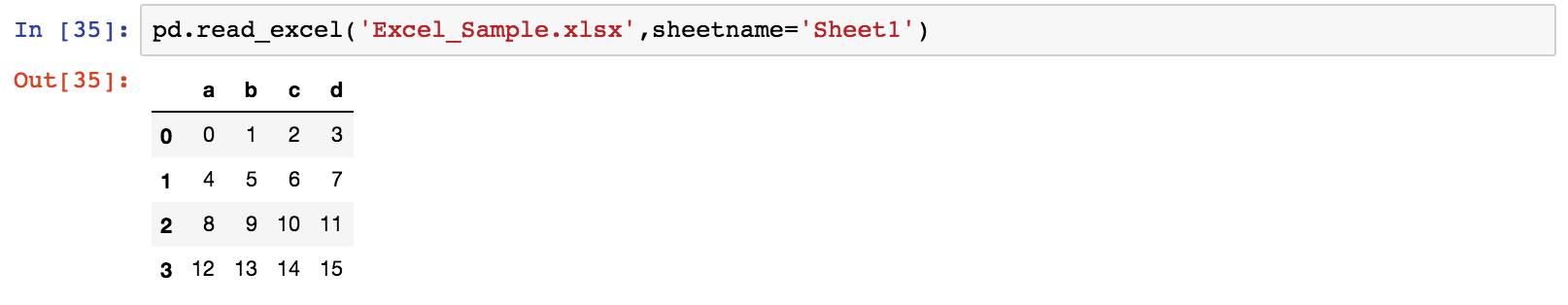
**CSV Output**

******

* 1. ***Excel***

Pandas can read and write excel files, keep in mind, this only imports data. Not formulas or images, having images or macros may cause this read\_excel method to crash.

**Excel Input**



**Excel Output**

******

* 1. ***HTML***

You may need to install htmllib5,lxml, and BeautifulSoup4. In your terminal/command prompt run:

*conda install lxml*

*conda install html5lib*

*conda install BeautifulSoup4*

Then restart Jupyter Notebook. (or use pip install if you aren't using the Anaconda Distribution)

Pandas can read table tabs off of html. For example:

**HTML Input**

Pandas read\_html function will read tables off of a webpage and return a list of DataFrame objects:



* 1. ***SQL***

The pandas.io.sql module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. Examples of such drivers are psycopg2 for PostgreSQL or pymysql for MySQL. For SQLite this is included in Python’s standard library by default. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

See also some cookbook examples for some advanced strategies.

The key functions are:

*read\_sql\_table(table\_name, con[, schema, ...])*

Read SQL database table into a DataFrame.

*read\_sql\_query(sql, con[, index\_col, ...])*

Read SQL query into a DataFrame.

*read\_sql(sql, con[, index\_col, ...])*

Read SQL query or database table into a DataFrame.

*DataFrame.to\_sql(name, con[, flavor, ...])*

Write records stored in a DataFrame to a SQL database.

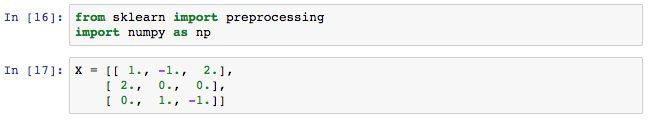
******

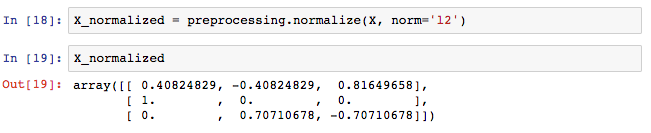
## **Data Nornalization**

Normalization is the process of scaling individual samples to have unit norm. This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples.

This assumption is the base of the Vector Space Model often used in text classification and clustering contexts.

The function normalize provides a quick and easy way to perform this operation on a single array-like dataset, either using the l1 or l2 norms:





The preprocessing module further provides a utility class Normalizer that implements the same operation using the Transformer API (even though the fit method is useless in this case: the class is stateless as this operation treats samples independently).

This class is hence suitable for use in the early steps of a sklearn.pipeline.Pipeline:

