Math 5680: Mathematics of Machine Learning

Lab 1: Pandas (Part I)

What is Pandas?

From https://pandas.pydata.org/pandas-docs/stable:

> Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language. It is already well on its way toward this goal.
>
> Pandas is well suited for many different kinds of data:
>
> - Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
> - Ordered and unordered (not necessarily fixed-frequency) time series data.
> - Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
> - Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

Pandas is built with NumPy

NumPy provides multi-dimensional list-like data structures which are **typed** and much faster than Python lists. The interface to the pandas data structures is very similar to the one provided by NumPy. In many cases the methods provided have the same, or similar, names. For more detailed discussion of NumPy, please refre to the documentation.

1. Importing Pandas

First, you need to `import pandas`. By convention, it is imported using the *alias* `pd`. To import using an alias use the following syntax:

```
import <library> as <alias>
```

- Many popular libraries try to define an alias convention, check their documentation

Task 1:

1. Try to import `pandas` using the alias convention?

In [1]:

```

```

2. Data Structures

Similar to the Python data structures (e.g. `list, dictionary, set`), Pandas provides three fundamental data structures:

1. `Series`: For one-dimensional data, similar to a Python list
2. `DataFrame`: For two-dimensional data, similar to a Python list of lists
3. `Index`: Similar to a `Series`, but for naming, selecting, and transforming data within a `Series` or `DataFrame`

2.1. Series

You can create a Pandas `Series` in a variety of ways, e.g.:

- From an assigned Python list:

In [2]:

```
a = ['a', 'b', 'c']
series = pd.Series(a)
series
```

Out[2]:

```
0    a
1    b
2    c
dtype: object
```

- From an unnamed Python list:

In [3]:

```
series = pd.Series([4, 5, 6])
series
```

Out[3]:
```
0    4
1    5
2    6
dtype: int64
```

- Using a specific index (similar to a `dict` where `index` are the keys):

In [4]:
```
series = pd.Series([4, 5, 6], index=["a", "b", "c"])
series
```

Out[4]:
```
a    4
b    5
c    6
dtype: int64
```

- Directly from a dictionary (exactly the same as above):

In [5]:
```
series = pd.Series({"a": 4, "b": 5, "c": 6})
series
```

Out[5]:
```
a    4
b    5
c    6
dtype: int64
```

2.2. DataFrame

This is the data structure that makes Pandas shine. A `DataFrame` is essentially a dictionary of `Series` objects. In a `DataFrame`, the `keys` map to `Series` objects which share a common `index`. We should start with an example:

In [6]:
```
rock_bands = ["Pink Floyd", "Rush", "Yes"]
year_formed = [1965, 1968, 1968]
location_formed = ["London, England", "Ontario, Canada", "London, England"]
df = pd.DataFrame({"year_formed": year_formed, "location_formed": location_fo
df
```

Out[6]:

	year_formed	location_formed
Pink Floyd	1965	London, England
Rush	1968	Ontario, Canada
Yes	1968	London, England

Breaking Down the Result

- The indicies are `"Pink Floyd"`, `"Rush"`, and `"Yes"`
- The keys to the DataFrame are `"year_formed"` and `"location_formed"`
- The lists are converted to `Series` objects which share the indices

This might not seem very powerful, except that `DataFrame` can be constructed from files! In the following cell, I will read a file `states.csv` and generate its statistics in two lines!

In [7]:
```
df = pd.read_csv("states.csv")
df.head(5)
```

Out[7]:

	State	Population (2016)	Population (2017)
0	Alabama	4860545	4874747
1	Alaska	741522	739795
2	Arizona	6908642	7016270
3	Arkansas	2988231	3004279
4	California	39296476	39536653

In [8]:
```
df.describe()
```

Out[8]:

	Population (2016)	Population (2017)
count	5.200000e+01	5.200000e+01
mean	6.284855e+06	6.328007e+06
std	7.197166e+06	7.257007e+06
min	5.849100e+05	5.793150e+05
25%	1.791484e+06	1.791128e+06
50%	4.261051e+06	4.298482e+06
75%	7.001715e+06	7.113638e+06
max	3.929648e+07	3.953665e+07

Task 2:

```
Use `pd.read_csv` to read in the csv file: `example.csv`
- It does not contain a header (add `header=None` to the
arguments)
- When working with a single dataframe it is assigned to the
name `df`, by convention
- The file is bar separated (add `sep='|'` to the arguments)
- Lastly set the column names (add `names=["First", "Second"]`)
```

In [9]:

```

```

Out[9]:

	1|2
0	3|4
1	5|6
2	7|8

In [10]:

```

```

Out[10]:

	0
0	1|2
1	3|4
2	5|6
3	7|8

In [11]:

```

```

Out[11]:

	First	Second
0	1	2
1	3	4
2	5	6
3	7	8

2.3. Viewing DataFrames

Jupyter has built in support for viewing `DataFrame` objects in a nice format. Example:

In [12]:

```
import pandas as pd
df = pd.DataFrame([0, 1, 2], index=[5, 6, 7], columns=["Example"])
df
```

Out[12]:

	Example
5	0
6	1
7	2

The result should have been a nice looking table. Reminders:

- The above `DataFrame` contains a single `Series` with the key `Example`
- The indices are on the left (in bold)
- The values are in columns underneath the key

If you only want to view a subset of the DataFrame, you can use the syntax `<df>.head()`. By default it will print only 5 rows from the top of your DataFrame. This is very useful when trying to view the *shape* of your data. You can print fewer rows by adding `n=<number>` to the arguments of `head`.

Task 3:

1. Run the definitions cell below
2. Print the DataFrame in the following ways:
   - Using the built in Jupyter view
   - The head
   - The first row

In [13]:

```
l = list(range(10))
df = pd.DataFrame({"a": l, "b": l, "c": l})
```

In [14]:

```
```

Out[14]:

	a	b	c
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7
8	8	8	8
9	9	9	9

In [15]:

```
```

Out[15]:

	a	b	c
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

In [16]:

```

```

Out[16]:

	a	b	c
0	0	0	0

2.4. Access and Types

You can access individual `Series` from `DataFrame` using two syntax:

- Like a dictionary: `<df>["<key>"]`
- Like a data member, `<df>.<key>`

Important notes about the data member style:

- doesn't support keys with spaces
- can't be used to assign values to a non-existent key

For these reasons, I tend to prefer the dictionary style for everything. You will see both styles in this document simply to increase your familiarity with both, but it is important to know the limitations.

If you want to know the types of your `DataFrame` 's `Series` s using `<df>.dtypes`

Task 4:

1. Run the definitions cell below
2. Access the `b` Series of `df` using both accessor syntax

- Why are two columns printed?
- What is the type of `df["b"]` ?
- What are the `dtypes` of `df` ?

In [17]:

```
df = pd.DataFrame({"a": [0, 1, 2], "b": [0.0, 1.0, 2.0], "c": ["pandas", "is"
df
```

Out[17]:

	a	b	c
0	0	0.0	pandas
1	1	1.0	is
2	2	2.0	great

In [18]:

```

```

Out[18]:

```
0    0.0
1    1.0
2    2.0
Name: b, dtype: float64
```

In [19]:

```

```

Out[19]:

```
0    0.0
1    1.0
2    2.0
Name: b, dtype: float64
```

In [20]:

```

```

Out[20]:

```
a      int64
b    float64
c     object
dtype: object
```

2.5. Slicing and Indexing

There are many ways to slice and dice DataFrames. Let's start with the least flexible option, selecting multiple columns. Let's make a new DataFrame in the following cell.

In [21]:

```
example = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6], "c": [7, 8, 9]})
example
```

Out[21]:

	a	b	c
0	1	4	7
1	2	5	8
2	3	6	9

To slice columns `a` and `c` we'll use a similar syntax to dictionary access, shown before, but instead we will ask for a list of columns instead of a single one, e.g.

In [22]:

```
example[["a", "c"]]
```

Out[22]:

	a	c
0	1	7
1	2	8
2	3	9

One can also slice rows using the `list` slicing syntax. Note you are **required** to specify a slice (something containing '`:`'). For example,

In [23]:

```
# zeroth row only
example[0:1]
```

Out[23]:

	a	b	c
0	1	4	7

In [24]:

```
# first row to end
example[1:]
```

Out[24]:

	a	b	c
1	2	5	8
2	3	6	9

In [25]:

```
# every other row
example[::2]
```

Out[25]:

	a	b	c
0	1	4	7
2	3	6	9

In [ ]:

```
# this will fail with `KeyError`
# -> remember this is dictionary style access and `0` isn't a key!
example[0]
```

2.6. More Complicated Access Patterns

You can narrow down rows and columns using `loc`, some examples:

In [27]:

```
# only row 1, columns 'a' and 'c'
example.loc[1:1, ["a", "c"]]
```

Out[27]:

	a	c
1	2	8

In [28]:

```
# all rows, columns 'a' to 'b'
example.loc[:, "a":"b"]
```

Out[28]:

	a	b
0	1	4
1	2	5
2	3	6

In [29]:

```
# single row, single column
example.loc[0, "a"]
```

Out[29]:

```
1
```

Task 5:

Using `loc` and the `example` DataFrame,

1. Run the definitions cell below
2. Try to print every other row
3. Try to print columns `b` to `c`
4. Try to print all columns of the final row

In [ ]:

```
example = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6], "c": [7, 8, 9]})
example
```

In [30]:

```
```

Out[30]:

	a	b	c
0	1	4	7
2	3	6	9

In [31]:

```
```

Out[31]:

	b	c
0	4	7
1	5	8
2	6	9

In [32]:

```

```

Out[32]:

	a	b	c
2	3	6	9

Note: `loc` is all about index/key access, what if the indices are characters? Run the following cell and then complete the Task 6.

In [34]:

```
example2 = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6], "c": [7, 8, 9]}, ind
example2.head()
```

Out[34]:

	a	b	c
A	1	4	7
B	2	5	8
C	3	6	9

Task 6:

Use `loc` and DataFrame `example2`, to

- Print rows `B` to `C` and columns `a` to `b`.
- What happens if you try to access the index numerically?

In [35]:

```

```

Out[35]:

	b	c
B	5	8
C	6	9

Note: To access `example2` w/ numerical indices, we need `iloc`.

Task 7:

1. Using `iloc` and `example2`, get rows `B` to `C` and columns `a` to `b`.

In [36]:

```

```

Out[36]:

	a	b
B	2	5
C	3	6

Note: You can also use the `list` style access I showed before, e.g.

In [37]:

```
example2.iloc[[1, 2], [0, 1]]
```

Out[37]:

	a	b
B	2	5
C	3	6

3. Built-in Statistics

Coming back to the original example:

In [38]:

```
states = pd.read_csv("states.csv", index_col=0)
states.head()
```

Out[38]:

	Population (2016)	Population (2017)
State		
Alabama	4860545	4874747
Alaska	741522	739795
Arizona	6908642	7016270
Arkansas	2988231	3004279
California	39296476	39536653

- One can easily access the statistics of the entire `DataFrame`

In [39]:

```
states.describe()
```

Out[39]:

	Population (2016)	Population (2017)
count	5.200000e+01	5.200000e+01
mean	6.284855e+06	6.328007e+06
std	7.197166e+06	7.257007e+06
min	5.849100e+05	5.793150e+05
25%	1.791484e+06	1.791128e+06
50%	4.261051e+06	4.298482e+06
75%	7.001715e+06	7.113638e+06
max	3.929648e+07	3.953665e+07

- There are 52 states according to the `count`. The `mean` population is about 6.3 million people for 2016 and 2017
- It is also possible to down select the statistics, e.g. if I want the mean for the key `Population (2016)`

In [40]:

```
states["Population (2016)"].mean()
```

Out[40]:

```
6284854.903846154
```

Task 8:

- Find the state with
  - the minimum (`min`) population in 2016
  - the maximum (`max`) population in 2017

In [41]:

```
```

Out[41]:

```
584910
```

In [42]:

```
```

Out[42]:

```
39536653
```

3.1. Adding New Columns

How would we find the average population *per state* for 2016 and 2017?

- We can use a dispatched operation similar to the `==` example previous to generate the averages

In [43]:

```
(states["Population (2016)"] + states["Population (2017)"]) / 2
```

Out[43]:
```
State
Alabama                  4867646.0
Alaska                    740658.5
Arizona                  6962456.0
Arkansas                 2996255.0
California              39416564.5
Colorado                 5568629.5
Connecticut              3587934.5
Delaware                  957318.5
District of Columbia      689154.0
Florida                 20820494.5
Georgia                 10371499.5
Hawaii                   1428110.5
Idaho                    1698484.5
Illinois                12818874.5
Indiana                  6650412.5
Iowa                     3138290.0
Kansas                   2910427.0
Kentucky                 4445151.0
Louisiana                4685245.0
Maine                    1333069.5
Maryland                 6038464.5
Massachusetts            6841770.0
Michigan                 9947878.0
Minnesota                5550828.0
Mississippi              2984757.5
Missouri                 6102354.0
Montana                  1044574.5
Nebraska                 1913839.5
Nevada                   2968646.5
New Hampshire            1338905.0
New Jersey               8992030.0
New Mexico               2086751.0
New York                19842842.5
North Carolina          10215054.0
North Dakota              755470.5
Ohio                    11640581.5
Oklahoma                 3926035.5
Oregon                   4114382.5
Pennsylvania            12796311.0
Rhode Island             1058602.5
South Carolina           4992095.5
South Dakota              865604.0
Tennessee                6682694.0
Texas                   28104729.0
Utah                     3073077.0
Vermont                   623505.5
Virginia                 8442200.0
Washington               7343338.5
West Virginia            1822247.0
Wisconsin                5784200.0
Wyoming                   582112.5
Puerto Rico              3371848.5
dtype: float64
```

- The above is a `Series` object. We can assign it to a `key` in the `DataFrame`

In [44]:
```
states["Average Population"] = (states["Population (2016)"] + states["Populat
states["Average Population"].head()
```

Out[44]:
```
State
Alabama         4867646.0
Alaska           740658.5
Arizona         6962456.0
Arkansas        2996255.0
California     39416564.5
Name: Average Population, dtype: float64
```

- Finally the overall mean

In [45]:
```
states["Average Population"].mean()
```

Out[45]:
```
6306430.865384615
```

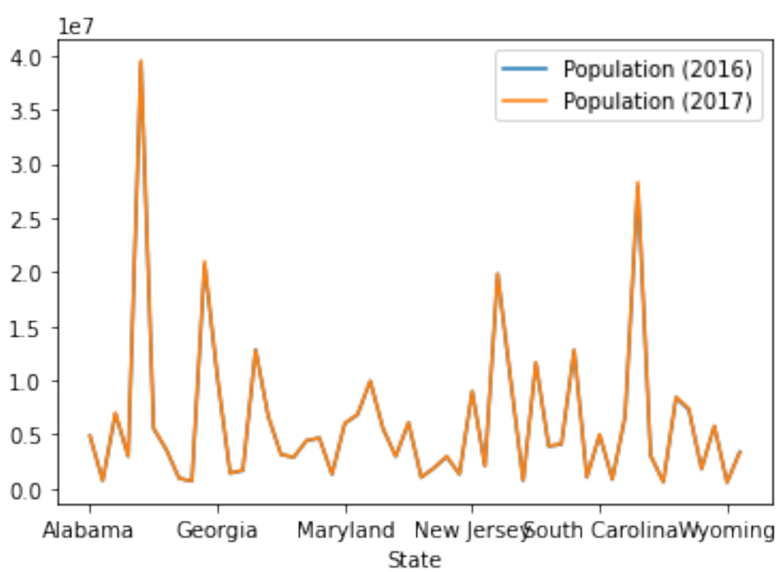
4. Viewing Data

Pandas plugs into `matplotlib` very nicely. I am going to iteratively build a plot which is easy to read. First, run the following cell.

In [46]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [47]:
```
states = pd.read_csv("states.csv", index_col=0)
states.plot()
```

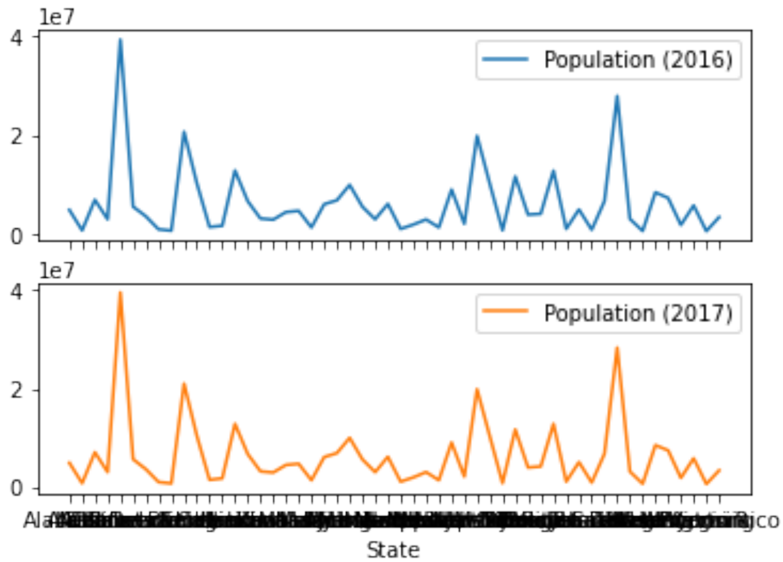
Out[47]:
```
<AxesSubplot:xlabel='State'>
```



This is something, but not very helpful. What would we like:

- X axis should be labeled with the state

In [48]:
```
ax = states.plot(subplots=True, xticks=range(states.shape[0]))
```

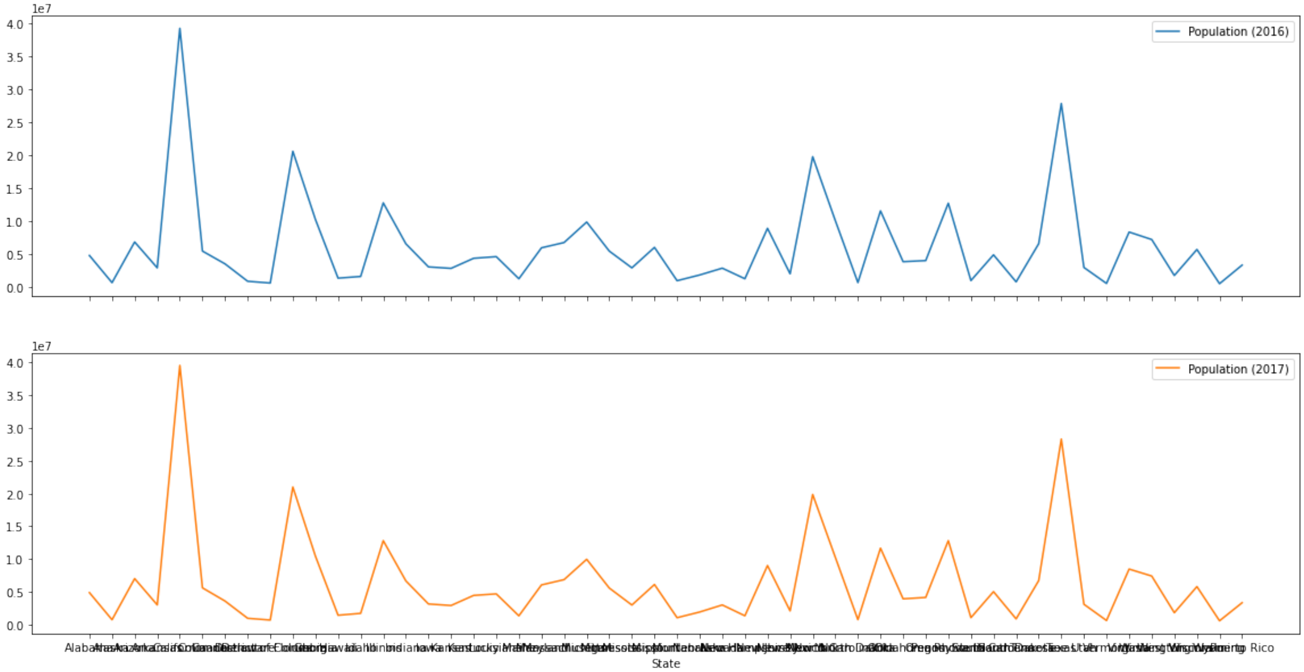


Notes:

1. `subplots=True` : separates the 2 plots from one another
2. `xticks=range(states.shape[0])` : sets all of the ticks on the x-axis
3. `ax = ...` : is a list containing both plots
4. `ax[0].set_xticklables` changes the numeric index to the State name, should only be necessary for the 0th plot
5. `suppressing_output = ...` , I use this to supress the output from `set_xticklabels`

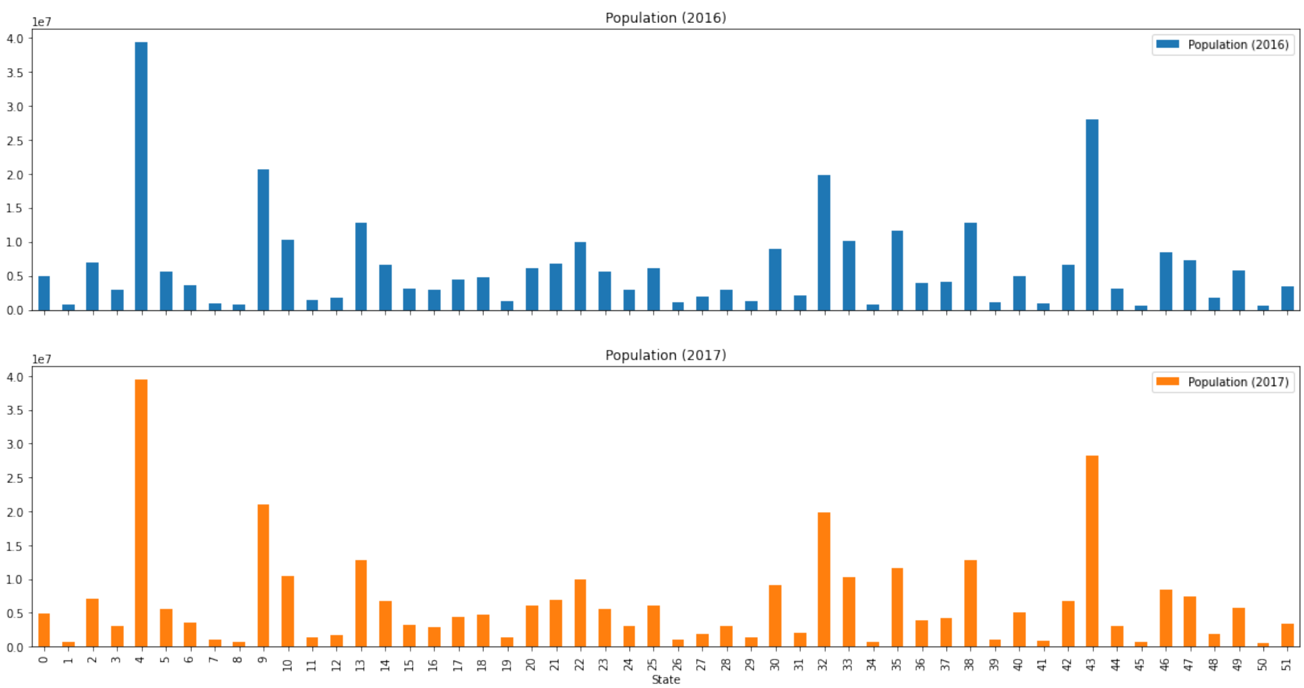
Neat, but I can't read the labels...

In [49]:
```
ax = states.plot(subplots=True, xticks=range(states.shape[0]), figsize=(20, 1
```



- The line plots are a little awkward because the points aren't connected in anyway

In [50]:
```
ax = states.plot(subplots=True, xticks=range(states.shape[0]), figsize=(20, 1
```



4.1. Apply + Lambda

This is a contrived example, but it shows the utility of `apply` + `lambda`. What if we wanted wanted to figure out if all letters A-Z are in the names of the states? First, we could create a `set` of characters in each state's name:

In [51]:

```
# don't use the names of states an the index!
states = pd.read_csv("states.csv")

def set_of_chars(s):
    return set(list(s.lower()))

series_of_sets = states.State.apply(lambda s: set_of_chars(s))
series_of_sets
```

```
Out[51]: 0                                   {l, a, m, b}
         1                                   {l, k, a, s}
         2                             {i, o, n, r, z, a}
         3                                {k, n, r, s, a}
         4                       {i, c, a, o, l, n, r, f}
         5                             {c, d, o, l, r, a}
         6                          {i, c, t, e, o, n, u}
         7                             {e, w, d, l, r, a}
         8     {i, c, a, t, m, d, o,  , l, r, u, s, f, b}
         9                          {i, a, d, o, l, r, f}
         10                            {i, e, o, r, g, a}
         11                                  {w, h, a, i}
         12                               {i, d, o, h, a}
         13                               {i, o, l, n, s}
         14                                  {n, i, a, d}
         15                                  {i, w, a, o}
         16                                  {n, k, a, s}
         17                         {c, k, t, e, n, u, y}
         18                         {i, o, l, n, u, s, a}
         19                               {i, m, e, n, a}
         20                         {m, d, l, n, r, y, a}
         21                      {c, t, m, e, u, s, h, a}
         22                         {i, c, m, n, g, h, a}
         23                      {i, t, m, e, o, n, s, a}
         24                                  {i, s, p, m}
         25                            {i, m, o, r, u, s}
         26                               {t, m, o, n, a}
         27                         {k, e, n, r, s, a, b}
         28                               {v, e, d, n, a}
         29                {i, m, e, w,  , n, r, s, p, h, a}
         30                      {e, w, j,  , n, r, s, y}
         31                   {i, c, m, e, w, o,  , n, x}
         32                      {k, e, w, o,  , n, r, y}
         33                {i, c, t, o,  , n, l, r, h, a}
         34                   {k, t, d, o,  , n, r, h, a}
         35                                     {i, h, o}
         36                            {k, m, o, l, h, a}
         37                               {e, o, n, r, g}
         38                   {i, v, e, n, l, s, p, y, a}
         39             {i, e, d, o,  , l, n, r, s, h, a}
         40          {i, c, t, o,  , l, n, r, u, s, h, a}
         41                   {k, t, d, o,  , u, s, h, a}
         42                                  {n, e, s, t}
         43                               {t, e, s, x, a}
         44                                  {u, a, t, h}
         45                         {t, m, v, e, o, n, r}
         46                            {i, v, n, r, g, a}
         47                   {i, t, w, o, n, s, g, h, a}
         48             {i, t, v, w, e,  , n, r, s, g, a}
         49                            {i, c, w, o, n, s}
         50                         {i, m, w, o, n, g, y}
         51                   {i, c, t, e, o,  , r, u, p}
         Name: State, dtype: object
```

Reminder: Lambdas

Reminder, a *lambda* constructs an ephemeral unnamed function. This is opposed to the named function `set_of_chars` above. The point is the `apply` method takes a function. We could have done the following:

```
series_of_sets = states.State.apply(lambda s:
set(list(s.lower())))
```

Or, simply:

```
series_of_sets = states.State.apply(set_of_chars)
```