



Creating Pivot Tables in Pandas - Lab

Introduction

In this lab, we'll learn how to make use of our newfound knowledge of pivot tables to work with real-world data. We'll start by exploring

Objectives

- Understand what *Long* vs. *Wide* means, and why the *Long* format is often considered "tidy".
- Understand what a multi-level hierarchical index is
- Use `.groupby` statements with aggregate functions
- Create Pivot tables with pandas
- Understand how to switch between *Long* and *Wide* using `stack()` and `unstack()`

Getting Started

Import Pandas and Matplotlib.pyplot Using Standard Aliases

In the cell below:

- Import `pandas` and set the standard alias
- Import `matplotlib.pyplot` and set the standard alias
- Run the `ipython` magic command to display matplotlib graphs inline within the notebook

```
In [1]: import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Load the Data

The data for this activity is stored in a file called `'causes_of_death.tsv'` which is a somewhat morbid dataset from the center for disease control. Note that the file extension `.tsv` indicates that this data is formatted slightly differently than the standard `.csv`, the difference being that it has 'tab separated values' instead of 'comma separated values'. As such, pass in the optional parameter `delimiter='\t'` into the `pd.read_csv()` method.

```
In [2]: df = pd.read_csv('causes_of_death.tsv', delimiter='\t')
```

Now, display the head of the DataFrame to ensure everything loaded correctly.

```
In [3]: df.head()
```

	Notes	State	State Code	Ten-Year Age Groups	Ten-Year Age Groups Code	Gender	Gender Code	Race	Race Code	Deaths	Population	Crude Rate
0	NaN	Alabama	1	< 1 year	1	Female	F	American Indian or Alaska Native	1002-5	14	3579	Unreliable
1	NaN	Alabama	1	< 1 year	1	Female	F	Asian or Pacific Islander	A-PI	24	7443	322.5
2	NaN	Alabama	1	< 1 year	1	Female	F	Black or African American	2054-5	2093	169339	1236.0
3	NaN	Alabama	1	< 1 year	1	Female	F	White	2106-3	2144	347921	616.2
4	NaN	Alabama	1	< 1 year	1	Male	M	Asian or Pacific Islander	A-PI	33	7366	448.0

Our data is currently in *Wide* format. We can tidy this up by converting it to *Long* format by using groupby statements to aggregate our data into a much neater, more readable format.

Groupby Aggregations

Complete the following groupby statements.

1) Groupby State and Gender. Sum the values.

```
In [4]: # Your code here
df.groupby(['State', 'Gender']).sum().head()
```

	Notes	State Code	Deaths
State	Gender		
Alabama	Female	0.0	40 430133
	Male	0.0	41 430647
Alaska	Female	0.0	80 27199
	Male	0.0	84 36135
Arizona	Female	0.0	180 396028

2) Groupby State and Gender and Race. Find the average values.

```
In [5]: # Your code here
df.groupby(['State', 'Gender', 'Race']).mean().head()
```

	Notes	State Code	Deaths
State	Gender	Race	

		American Indian or Alaska Native	NaN	1.0	70.875000
Alabama	Female	Asian or Pacific Islander	NaN	1.0	95.500000
		Black or African American	NaN	1.0	9074.000000
		White	NaN	1.0	29890.636364
		Male	American Indian or Alaska Native	NaN	1.0
					86.375000

3) Groupby Gender and Race. Find the minimum values.

In [6]: # Your code here
df.groupby(['Gender', 'Race']).min().head()

Out[6]:	Notes	State	State Code	Ten-Year Age Groups	Ten-Year Age Groups Code	Gender Code	Race Code	Deaths	Population	Crude Rate
Gender	Race									
Female	American Indian or Alaska Native	NaN	Alabama	1	1-4 years	1	F	1002-5	10	100285
	Asian or Pacific Islander	NaN	Alabama	1	1-4 years	1	F	A-PI	10	100371
	Black or African American	NaN	Alabama	1	1-4 years	1	F	2054-5	10	100170
	White	NaN	Alabama	1	1-4 years	1	F	2106-3	12	1000704
Male	American Indian or Alaska Native	NaN	Alabama	1	1-4 years	1	M	1002-5	10	1001
										10000.0

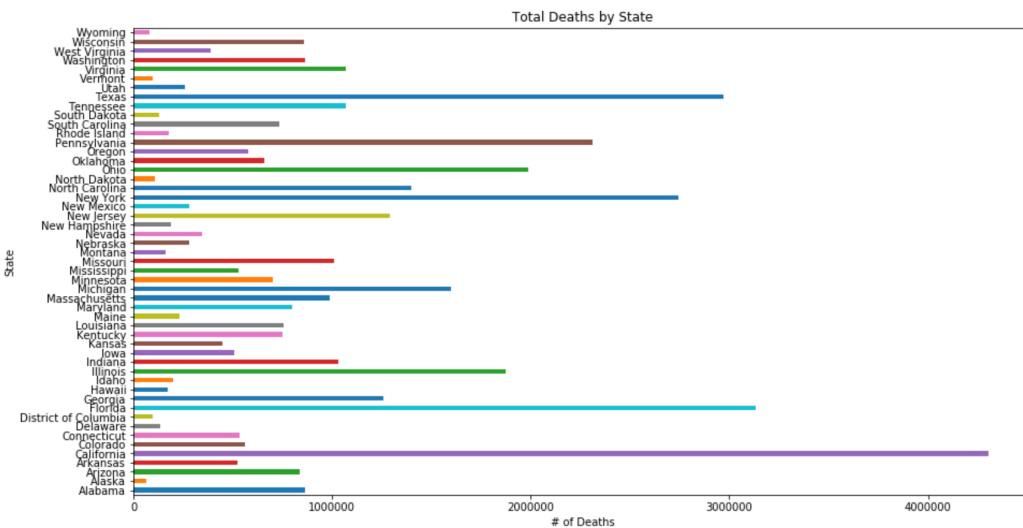
4) Create a bar chart of the total number of deaths by state.

- Sort your columns in order (ascending or descending are both acceptable).
- Also make sure to include a title, axes labels and have your graph be an appropriate size.

NOTE: In order to do this, slice the Deaths column after the .groupby() method, but before the sum() method. You can even chain the .plot() call on after the sum() call and do this all on one line, excluding the labeling of the graph!

In [7]: #Your code here
df.groupby(['State'])['Deaths'].sum().plot(kind='barh', figsize=(15,8))
plt.title('Total Deaths by State')
plt.xlabel("# of Deaths")

Out[7]: Text(0.5,0,'# of Deaths')



Inspecting our Data

Let's go one step further and get a print-out of the data type of each column.

In the cell below, get the .info() of our DataFrame, and note the data type that each column is currently stored as.

In [8]: df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4115 entries, 0 to 4114
Data columns (total 12 columns):
Notes          0 non-null float64
State           4115 non-null object
State Code      4115 non-null int64
Ten-Year Age Groups 4115 non-null object
Ten-Year Age Groups Code 4115 non-null object
Gender          4115 non-null object
Gender Code     4115 non-null object
Race            4115 non-null object
Race Code       4115 non-null object
Deaths          4115 non-null int64
Population      4115 non-null object
Crude Rate      4115 non-null object
dtypes: float64(1), int64(2), object(9)
memory usage: 385.9+ KB
```

Let's look at some samples from the Population column to see if the current encoding seems appropriate for the data it contains.

In the cell below, display the population values for the first 5 rows in the DataFrame.

To [41]: df.Population.iloc[:5]

```
Out[9]: 0      3579
        1      7443
        2     169339
        3     347921
        4      7366
Name: Population, dtype: object
```

Just to be extra sure, let's check the value counts to see how many times each unique value shows up in the dataset. We'll only look at the top 5.

In the cell below, print out the top 5 `value_counts()` of the population column of the DataFrame.

```
In [10]: df.Population.value_counts()[:5]
```

```
Out[10]: Not Applicable    75  
        4258                 2  
        6420                 2  
        11680                2  
        5142                 2  
Name: Population, dtype: int64
```

Clearly, this data should be stored as a numeric type, not a categorical type.

5a) Reformat the Population Column as an Integer

As stands, not all values will be able to be reformatted as integers. Most of the cells in the the `Population` column contain integer values, but the entire column is currently encoded in string format because some cells contain the string "Not Applicable".

We need to remove these rows before we can cast the Population column to an Integer data type.

In the cell below:

- Slice the rows of `df` where the Population column is equal to 'Not Applicable'.
 - Use `to_drop.index` to drop the offending rows from `df`. Be sure to set the `axis=0`, and `inplace=True`.
 - Cast the Population column to an integer data type using the `.astype()` function, with the single parameter `int64` passed in.
 - Print the Population column's `dtype` attribute to confirm it is now stored in `int64` format.

NOTE: `.astype()` returns a copy of the column, so make sure you set the `Population` column equal to what this method returns--don't just call it!

```
In [11]: #Your code here
to_drop = df[df['Population'] == 'Not Applicable']
df.drop(to_drop.index, axis=0, inplace=True)
df.Population = df.Population.astype('int64')
print(df.Population.dtype)

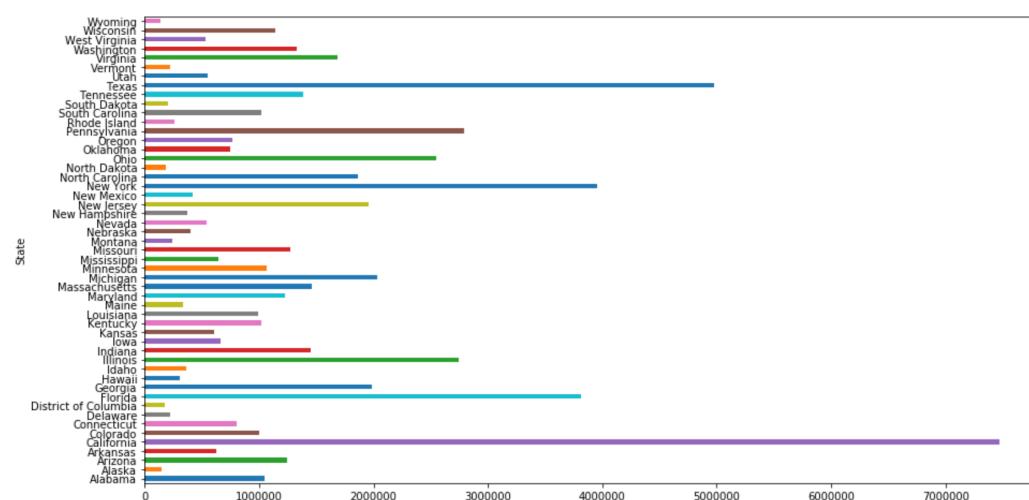
int64
```

5b) Complete the Bar Chart

Now that we've reformatted our data, let's create a bar chart of the Mean Population by State

```
In [12]: #Your code here
df.groupby('State')[['Population']].mean().plot(kind='barh', figsize=(15,8))

Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x200c215550>
```



Below we will investigate how we can combine the `pivot` method along with the `groupby` method to combine some cool `stacked bar charts`!

Using Aggregate Functions

In the cell below:

- Group `df` by 'State' and 'Gender', and then slice both 'Death' and 'Population' from it. Chain the `.agg()` method to return the mean, min, max, and standard deviation these sliced columns.

NOTE: This only requires one line of code

By now, you've probably caught on that the code required to do this follows this pattern: ([things to group by])[columns to slice].agg([aggregates to return]).

Then, display the head of this of this new DataFrame.

```
In [13]: # A sample groupby similar to above.  
grouped = df.groupby(['State', 'Gender'])[['Deaths',
```

```
grouped.head()
```

Out[13]:

Note how Pandas denotes a multi-hierarchical index in the DataFrame above

Let's inspect how a multi-hierarchical index is actually stored

In the cell below, display the `index` attribute of this DataFrame

```
In [14]: grouped.index
```

A two-dimensional array denotes the multiple levels, with each possible combination being a row in our grouped DataFrame.

Let's reset the index, and then see how it changes.

In the cell below, call the DataFrame's `reset_index()` function. Then, display the head of the DataFrame.

```
In [15]: # First, reset the index. Notice the subtle difference; State and Gender are now columns rather than the index.
grouped = grouped.reset_index()
grouped.head()
```

Out[15]:	State	Gender	Deaths						Population		
			mean	min	max	std	mean	min	max	std	
0	Alabama	Female	10753.325000	10	116297	24612.250487	1.078713e+06	2087	4334752	1.400309e+06	
1	Alabama	Male	10765.850000	10	88930	20813.538537	1.014946e+06	1129	4284775	1.397830e+06	
2	Alaska	Female	679.975000	13	4727	1154.870455	1.440403e+05	1224	682855	2.015793e+05	
3	Alaska	Male	860.357143	12	5185	1411.777392	1.518884e+05	578	770502	2.238843e+05	
4	Arizona	Female	8998.386264	21	132923	26245.941003	1.246502e+06	12211	6265346	2.096632e+06	

Note how the way the index is displayed has changed. The index columns that made up the multi-hierarchical index before are now stored as columns of data, with each row given a more traditional numerical index.

Let's confirm this by reexamining the `index` attribute of `grouped` in the cell below.

In [16]: grouped index

```
Out[16]: RangeIndex(start=0, stop=102, step=1)
```

However, look again at the displayed DataFrame--specifically, the columns. Resetting the index has caused the DataFrame to use a multi-indexed structure for the columns.

In the cell below, examine the `columns` attribute of `grouped` to confirm this.

```
In [17]: #Notice that this causes columns to be MultiIndexed!
grouped.columns
```

```
Out[17]: MultiIndex(levels=[[['Deaths', 'Population', 'Gender', 'State'], ['mean', 'min', 'max', 'std', '']],  
                           labels=[[3, 2, 0, 0, 0, 1, 1, 1, 1], [4, 4, 0, 1, 2, 3, 0, 1, 2, 3]]])
```

Column Levels

Since we're working with multi-hierarchical indices, we can examine the indices available at each level.

In the cell below, use the `get_level_values` method contained within the DataFrame's `columns` object to get the values for the outermost layer of the index.

```
In [18]: grouped.columns.get_level_values(0)
```

```
Out[18]: Index(['State', 'Gender', 'Deaths', 'Deaths', 'Deaths', 'Deaths', 'Population',
       'Population', 'Population', 'Population'],
      dtype='object')
```

Now, get the level values for the inner layer of the index

```
In [19]: grouped.columns.get_level_values(1)
```

```
Out[19]: Index(['', '', 'mean', 'min', 'max', 'std', 'mean', 'min', 'max', 'std'], dtype='object')
```

Flattening the DataFrame

We can also *flatten* the DataFrame from a multi-hierarchical index to more traditional one-dimensional index. We do this by creating each unique combination possible of every level of the multi-hierarchical index. Since this is a complex task, you do not need to write it, but take some time to examine the code in the `get_flattened_index` function.

possible at every level of the multi-level hierarchical index. Since this is a complex task, you do not need to write it—but take some time to examine the code in the cell below and see if you can understand how it works!

```
In [20]: #We could also flatten these:  
cols0 = grouped.columns.get_level_values(0)  
cols1 = grouped.columns.get_level_values(1)  
grouped.columns = [col + '_' + col1 if col1 != '' else col0 for col0, col1 in list(zip(cols0, cols1))]  
#The List comprehension above is more complicated than what we need but creates a nicer formatting and  
#demonstrates using a conditional within a List comprehension.  
#This simpler version works but has some tail underscores where col1 is blank:  
#grouped.columns = [col0 + '_' + col1 for col0, col1 in list(zip(cols0, cols1))]  
grouped.columns
```

```
Out[20]: Index(['State', 'Gender', 'Deaths_mean', 'Deaths_min', 'Deaths_max',  
               'Deaths_std', 'Population_mean', 'Population_min', 'Population_max',  
               'Population_std'],  
              dtype='object')
```

Now that we've flattened the DataFrame, let's inspect a couple rows to see what it looks like.

In the cell below, inspect the head of the `grouped` DataFrame.

```
In [21]: grouped.head()
```

```
Out[21]:   State  Gender  Deaths_mean  Deaths_min  Deaths_max  Deaths_std  Population_mean  Population_min  Population_max  Population_std  
0  Alabama  Female    10753.325000       10    116297  24612.250487  1.078713e+06      2087    4334752  1.400309e+06  
1  Alabama   Male     10765.850000       10    88930  20813.538537  1.014946e+06     1129    4284775  1.397830e+06  
2  Alaska  Female    679.975000        13     4727  1154.870455  1.440403e+05     1224    682855  2.015793e+05  
3  Alaska   Male     860.357143        12     5185  1411.777392  1.518884e+05      578    770502  2.238843e+05  
4  Arizona  Female    8998.386364       21    133923  26245.941003  1.246502e+06     12211   6265346  2.096632e+06
```

Using Pivots

Now, we'll gain some practice using the DataFrame class's built-in `.pivot()` method.

In the cell below, call the DataFrame's pivot method with the following parameters:

- `index = 'State'`
- `columns = 'Gender'`
- `values = 'Deaths_mean'`

Then, display the head of our new `pivot` DataFrame to see what it looks like.

```
In [22]: # Now it's time to pivot!  
pivot = grouped.pivot(index='State', columns='Gender', values='Deaths_mean')  
pivot.head()
```

```
Out[22]:   Gender      Female      Male  
          State  
Alabama  10753.325000  10765.850000  
Alaska   679.975000   860.357143  
Arizona  8998.386364  10036.204545  
Arkansas 6621.615385  6301.690476  
California 48312.840909 49555.522727
```

Great! We've just created a pivot table.

Let's reset the index and see how it changes our pivot table.

In the cell below, reset the index of the `pivot` object as we did previously. Then, display the head of the object to see if we can detect any changes.

```
In [23]: # Again, notice the subtle difference of resetting the index:  
pivot = pivot.reset_index()  
pivot.head()
```

```
Out[23]:   Gender  State  Female  Male  
0  Alabama  10753.325000  10765.850000  
1  Alaska   679.975000   860.357143  
2  Arizona  8998.386364  10036.204545  
3  Arkansas 6621.615385  6301.690476  
4  California 48312.840909 49555.522727
```

Visualizing Data With Pivot Tables

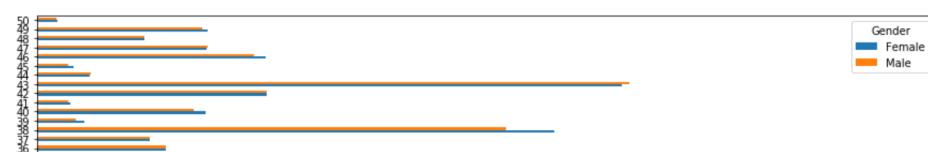
Now, we'll make use of our newly created pivot table to quickly create some visualizations of our data.

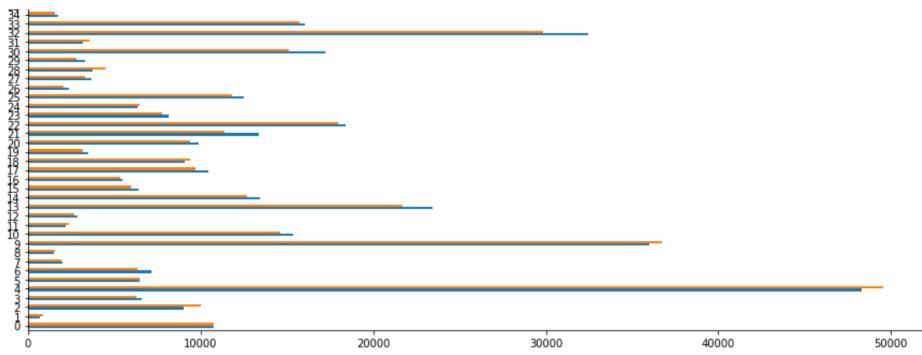
In the cell below, call `pivot.plot()` with the following parameters:

- `kind = 'barh'`
- `figsize = (15,8)`

```
In [24]: # Now Let's make a sweet bar chart!!  
pivot.plot(kind='barh', figsize=(15,8))
```

```
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x2193ca14be0>
```





Notice the Y-axis is currently just a list of numbers. That's because when we reset the index, it defaulted to assigning integers as the index for the DataFrame. Let's set the index back to 'State', and then recreate the visualization.

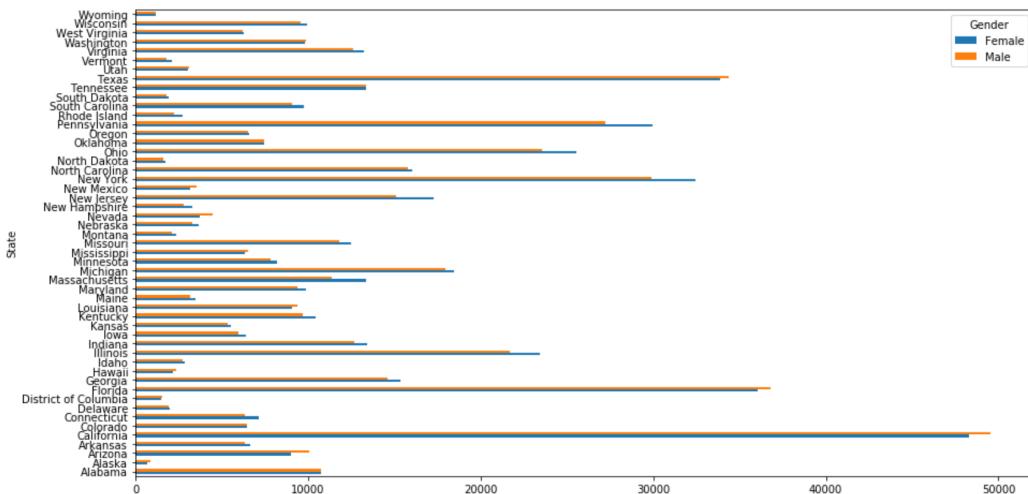
In the cell below:

- Use the `pivot` object's `set_index()` method and set the index to 'State'. Then, chain this with a `.plot()` call to recreate the visualization using the code we used in the cell above.

All the code in this cell should be done in a single line. Just call the methods--do not rebind `pivot` to be equal to this line of code.

```
In [25]: #Where's the states?! Notice the y-axis is just a list of numbers.
#This is populated by the DataFrame's index.
#When we used the .reset_index() method, we created a new numbered index to name each row.
#Let's fix that by making state the index again.
pivot.set_index('State').plot(kind='barh', figsize=(15,8))
```

```
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x2193ce8a470>
```



Now, that we've created a visualization with the states as the y-axis, let's print out the head of the `pivot` object again.

```
In [26]: # Also notice that if we call the DataFrame pivot again, state is not it's index.
#The above method returned a DataFrame with State as index and we plotted it,
#but it did not update the DataFrame itself.
pivot.head(2)
```

```
Out[26]: Gender    State   Female    Male
0   Alabama  10753.325 10765.850000
1   Alaska   679.975  860.357143
```

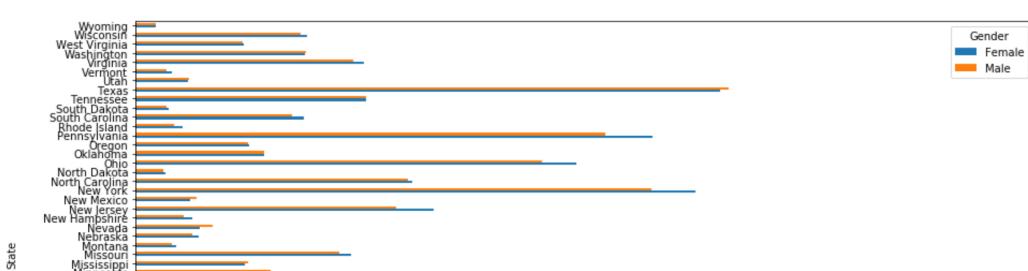
Note that the index has not changed. That's because the code we wrote when we set the index to the 'State' column returns a copy of the DataFrame object with the index set to 'State'--by default, it does not mutate original `pivot` object.

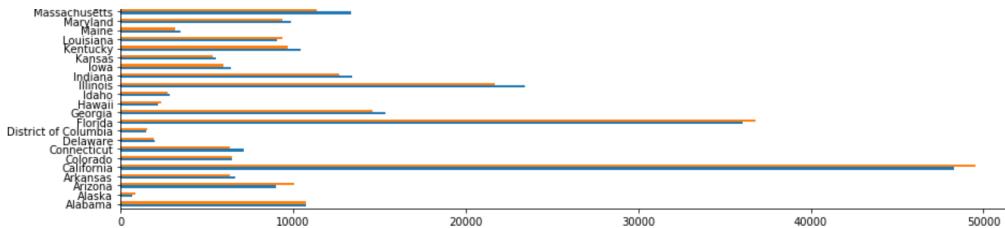
If we want to do that, we'll need to capture the new object returned by updating the contents of the `pivot` variable.

In the cell below, set the index of `pivot` to 'State'. Then, recreate the bar plot using this new object.

```
In [27]: #If we wanted to more permanently change the index we would set it first and then plot:
pivot = pivot.set_index('State')
pivot.plot(kind='barh', figsize=(15,8))
```

```
Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x2193cf55160>
```





Again, let's check the head of the DataFrame to confirm that the index structure has changed.

In [28]: `pivot.head(2)`

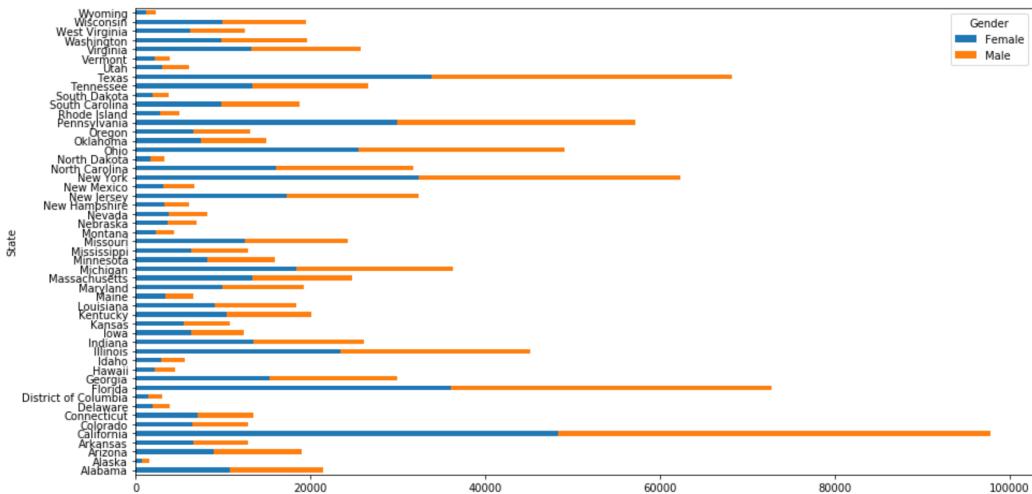
```
Out[28]:   Gender    Female      Male
          State
Alabama  10753.325  10765.850000
Alaska    679.975   860.357143
```

Finally, let's stack these bar charts to see how that looks.

In the cell below, recreate the visualization we did in the cell above, but this time, also pass in `stacked=True` as a parameter.

In [29]: `# Lastly, Let's stack each of these bars for each state.
#Notice we don't have to worry about index here, because we've already set it above.
pivot.plot(kind='barh', figsize=(15,8), stacked=True)`

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x2193cf29e48>



Stacking and Unstacking DataFrames

Now, let's get some practice stacking and unstacking DataFrames.

Stacking

In the cell below, let's display the head of `grouped` to remind ourselves of the format we left it in.

In [30]: `grouped.head()`

```
Out[30]:   State  Gender  Deaths_mean  Deaths_min  Deaths_max  Deaths_std  Population_mean  Population_min  Population_max  Population_std
0  Alabama  Female     10753.325       10     116297  24612.250487   1.078713e+06        2087     4334752  1.400309e+06
1  Alabama   Male     10765.850000      10     88930  20813.538537   1.014946e+06       1129     4284775  1.397830e+06
2  Alaska  Female      679.975       13      4727  1154.870455   1.440403e+05       1224      682855  2.015793e+05
3  Alaska   Male      860.357143      12      5185  1411.777392   1.518884e+05       578      770502  2.238643e+05
4  Arizona  Female     8998.386364      21     133923  26245.941003   1.246502e+06      12211     6265346  2.096632e+06
```

As we can see above, `grouped` is currently in a flattened format, with no hierarchical structure to its indices.

In the cell below, call the `grouped` DataFrame's `.stack()` method.

In [31]: `grouped.stack()`

```
Out[31]: 0  State      Alabama
          Gender    Female
          Deaths_mean  10753.3
          Deaths_min    10
          Deaths_max   116297
          Deaths_std   24612.3
          Population_mean  1.078713e+06
          Population_min  2087
          Population_max  4334752
          Population_std  1.400309e+06
1  State      Alabama
          Gender    Male
          Deaths_mean  10765.9
          Deaths_min    10
          Deaths_max   88930
          Deaths_std   20813.5
          Population_mean  1.014946e+06
          Population_min  1129
          Population_max  4284775
          Population_std  1.397830e+06
```

```

Population_mean    1.01495e+00
Population_min     1129
Population_max     4284775
Population_std    1.39783e+06
2      State          Alaska
Gender            Female
Deaths_mean       679.975
Deaths_min        13
Deaths_max        4727
Deaths_std        1154.87
Population_mean   144040
Population_min    1224
Population_max    682855
Population_std   201579
...
99      State          Wisconsin
Gender           Male
Deaths_mean      9573.45
Deaths_min       13
Deaths_max       113692
Deaths_std       25681.4
Population_mean  1.13532e+06
Population_min   1286
Population_max   6860107
Population_std   2.08907e+06
100     State          Wyoming
Gender           Female
Deaths_mean      1161.03
Deaths_min       10
Deaths_max       13140
Deaths_std       2937.94
Population_mean  146757
Population_min   336
Population_max   672620
Population_std   235238
101     State          Wyoming
Gender           Male
Deaths_mean      1149.51
Deaths_min       10
Deaths_max       10113
Deaths_std       2569.28
Population_mean  139224
Population_min   244
Population_max   694760
Population_std   241360
Length: 1020, dtype: object

```

As we can see, the `stack()` method has stacked our DataFrame from a flattened format into one with a multi-hierarchical index! This is an easy, quick way to aggregate our data.

Unstacking

Now, we'll explore unstacking with the `pivot` DataFrame, which is already stacked into a pivot table.

In the cell below, set `unstack` `pivot` using the object's `.unstack()` method. Then, display the object to see how it has changed.

```
In [32]: pivot = pivot.unstack()
pivot
```

```

Out[32]: Gender  State
Female  Alabama      10753.325000
          Alaska       679.975000
          Arizona      8998.386364
          Arkansas     6621.615385
          California    48312.840999
          Colorado      6460.162791
          Connecticut   7144.641026
          Delaware      2000.029412
          District of Columbia 1497.580645
          Florida       36019.071429
          Georgia       15372.317073
          Hawaii        2182.944444
          Idaho         2874.323529
          Illinois      23432.926829
          Indiana       13425.717949
          Iowa          6419.707317
          Kansas         5492.309524
          Kentucky      10426.083333
          Louisiana     9076.585366
          Maine          3471.823529
          Maryland       9894.780488
          Massachusetts 13356.846154
          Michigan       18421.659091
          Minnesota     8168.204545
          Mississippi    6342.634146
          Missouri      12493.170732
          Montana        2341.393939
          Nebraska       3667.794872
          Nevada         3729.166667
          New Hampshire 3293.344828
          ...
Male    Massachusetts  11368.341463
          Michigan      17940.431818
          Minnesota     7792.795455
          Mississippi    6487.317073
          Missouri      11810.119048
          Montana        2081.102564
          Nebraska       3290.682927
          Nevada         4489.261905
          New Hampshire 2800.303030
          New Jersey     15085.317073
          New Mexico      3549.428571
          New York        29864.477273
          North Carolina 15750.409091
          North Dakota   1587.411765
          Ohio            23551.951220
          Oklahoma        7468.909091
          Oregon          6528.977273
          Pennsylvania    27187.463415
          Rhode Island   2239.243243
          ...

```

```
south carolina      9877.47745  
South Dakota      1800.50000  
Tennessee        13333.05000  
Texas             34347.636364  
Utah              3081.511628  
Vermont           1785.846154  
Virginia          12585.833333  
Washington         9877.431818  
West Virginia     6211.612903  
Wisconsin         9573.454545  
Wyoming            1149.514286  
Length: 102, dtype: float64
```

Note that it has unstacked the multi-hierarchical structure of the `pivot` DataFrame by one level. Let's call it one more time and display the results!

In the cell below, set `pivot` equal to `pivot.unstack()` again, and then display the `pivot` object to see how things have changed.

```
In [33]: pivot = pivot.unstack()  
pivot
```

```
Out[33]:
```

	State	Alabama	Alaska	Arizona	Arkansas	California	Colorado	Connecticut	Delaware	District of Columbia	Florida	...	South Dakota
Gender													
Female	10753.325	679.975000	8998.386364	6621.615385	48312.840909	6460.162791	7144.641026	2000.029412	1497.580645	36019.071429	...	1932.757576	
Male	10765.850	860.357143	10036.204545	6301.690476	49555.522727	6442.500000	6315.300000	1940.914286	1534.806452	36771.340909	...	1800.500000	

2 rows × 51 columns

After calling `unstack` a second time, we can see that `pivot` has a flattened structure, since it has been completely unstacked!

Conclusion

In this lab, we learned how to:

- Use `groupby` to stack and slice data conditionally
- Use aggregate functions in combination with `groupby` statements
- Create pivot tables with pandas
- Leverage pivot tables and `groupby` statements to create quick visualizations
- stack and unstack DataFrames