learn-co-curriculum / **dsc-pivot-tables-pandas-lab** Public

View license

0 stars 201 forks

Star Watch

Code Issues Pull requests 2 Actions Projects Security Insights

solution

This branch is 8 commits ahead, 8 commits behind master. Contribute

sumedh10 Merge pull request #2 from learn-co-curriculum/v2-1 ... on Oct 17, 2019 10

View code

README.md

Pivot Tables with 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.

Objectives

In this lab you will:

- Describe what is meant by long and wide format data
- Use multi-hierarchical indexing to access aggregated data
- Use pivot to create a more organized aggregated DataFrame
- Use stack and unstack to move between different level of multi-indexing

Getting Started

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

```
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 then 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()` function.

```
df = pd.read_csv('causes_of_death.tsv', delimiter='\t')
```

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

```
df.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Notes	State	State Code	Ten-Year Age Groups	Ten-Year Age Groups Code	Gender	Gender Code	Rac
0	NaN	Alabama	1	< 1 year	1	Female	F	Ameri Indian Alaska Native
1	NaN	Alabama	1	< 1 year	1	Female	F	Asian Pacific Island
2	NaN	Alabama	1	< 1 year	1	Female	F	Black Africa Ameri
3	NaN	Alabama	1	< 1 year	1	Female	F	White
4	NaN	Alabama	1	< 1 year	1	Male	M	Asian Pacific Island

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.

- Groupby `State` and `Gender`. Sum the values.

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

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

		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

- Groupby `State` , `Gender` , and `Race` . Find the average values.

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

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

			Notes	State Code	Deaths
State	**Gender**	**Race**			

			Notes	State Code	Deaths
State	**Gender**	**Race**			
Alabama	**Female**	**American Indian or Alaska Native**	NaN	1.0	70.875000
		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

- Groupby `Gender` and `Race` . Find the minimum values.

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

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

		Notes	State	State Code	Ten-Year Age Groups	Ten-Year Age Groups Code	Gender Code
Gender	**Race**						

		Notes	State	State Code	Ten-Year Age Groups	Ten-Year Age Groups Code	Gender Code
Gender	**Race**						
Female	**American Indian or Alaska Native**	NaN	Alabama	1	1-4 years	1	F
	Asian or Pacific Islander	NaN	Alabama	1	1-4 years	1	F
	Black or African American	NaN	Alabama	1	1-4 years	1	F
	White	NaN	Alabama	1	1-4 years	1	F
Male	**American Indian or Alaska Native**	NaN	Alabama	1	1-4 years	1	M

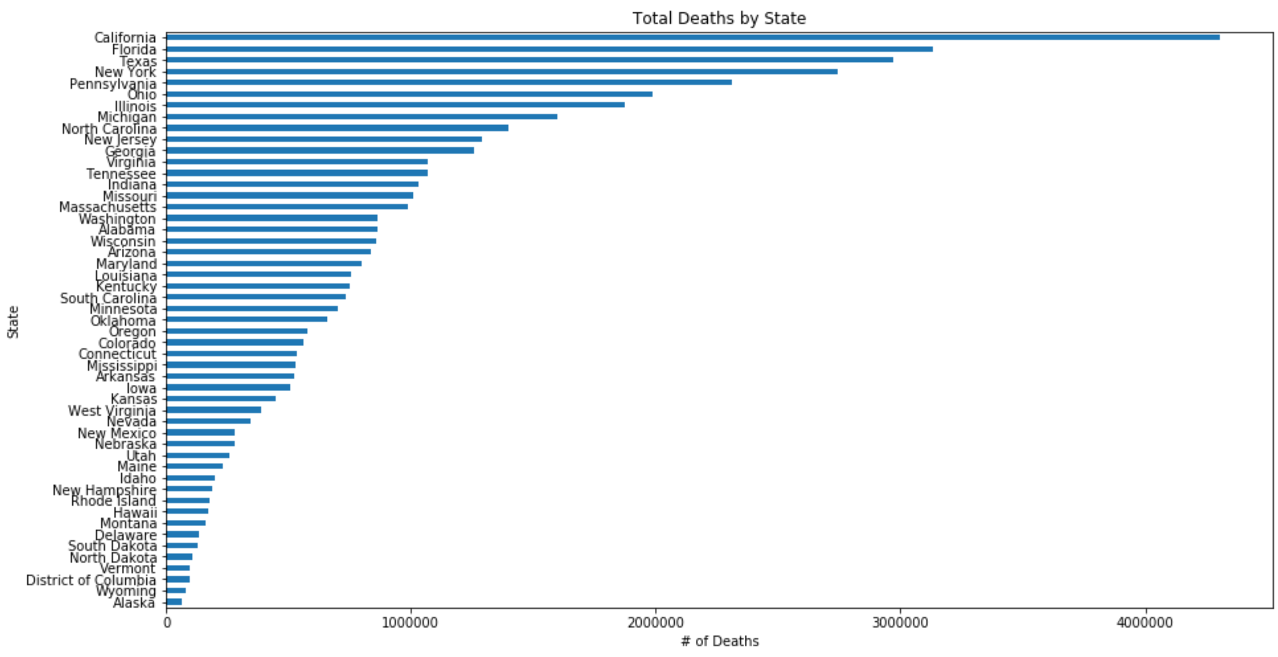
Create a bar chart of the total number of deaths by state:

- 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()` method on after the `.sum()` method and do this all on one line, excluding the labeling of the graph!

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

```
Text(0.5, 0, '# of Deaths')
```



Inspecting our data

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

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

```
df.info()
```

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.

```
df.Population.iloc[:5]
```

```
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.

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

```
Not Applicable    75
14810              2
113598             2
11680              2
6420               2
Name: Population, dtype: int64
```

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

Reformat the `Population` column as an integer

As it stands, not all values can be reformatted as integers. Most of the cells in 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()` method, 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!

```
# 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)
```

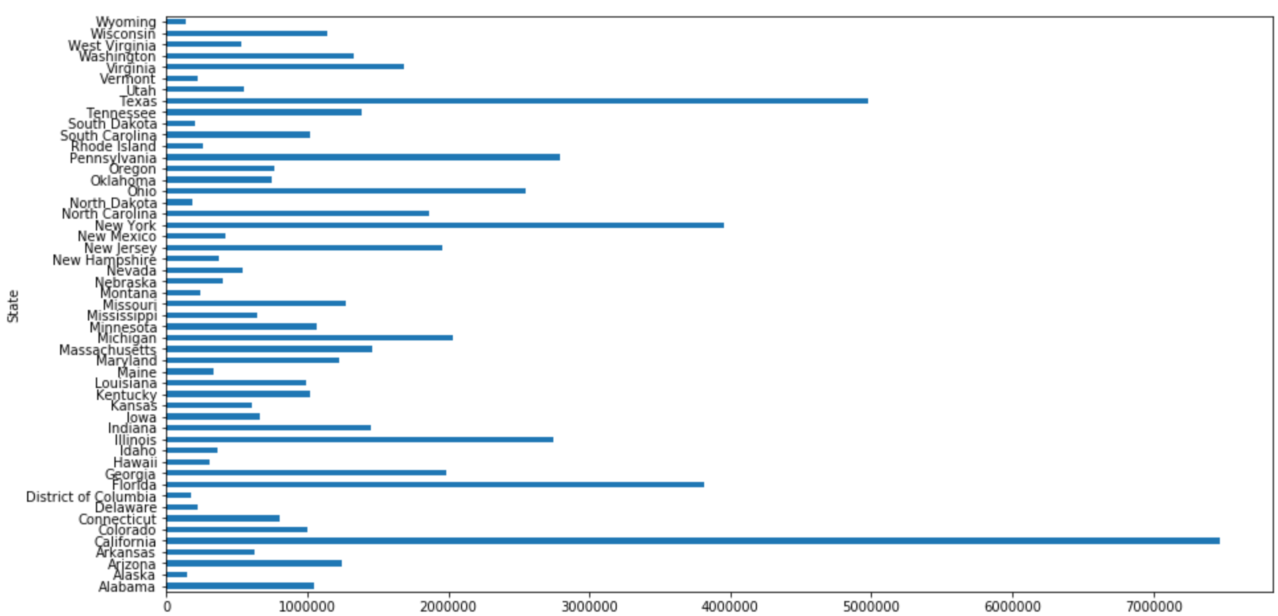
```
int64
```

Complete the bar chart

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

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

```
<matplotlib.axes._subplots.AxesSubplot at 0x120e7fa90>
```



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

Use aggregate methods

In the cell below:

- Group `df` by `'State'` and `'Gender'`, and then slice both `'Deaths'` and `'Population'` from it. Chain the `.agg()` method to return the mean, min, max, and standard deviation of 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 new DataFrame.

```
# Your code here
grouped = df.groupby(['State', 'Gender'])['Deaths',
                                          'Population'].agg(['mean',
                                                             'min', 'max', 'std'])
grouped.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead tr th {
    text-align: left;
}

.dataframe thead tr:last-of-type th {
    text-align: right;
}
```

</style>

		Deaths				
		mean	min	max	std	mean
State	Gender					
Alabama	Female	10753.325000	10	116297	24612.250487	1.078713e
	Male	10765.850000	10	88930	20813.538537	1.014946e
Alaska	Female	679.975000	13	4727	1154.870455	1.440403e
	Male	860.357143	12	5185	1411.777392	1.518884e

		Deaths				
		mean	min	max	std	mean
State	Gender					
Arizona	Female	8998.386364	21	133923	26245.941003	1.246502e

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.

```
grouped.index
```

```
MultiIndex(levels=[['Alabama', 'Alaska', 'Arizona', 'Arkansas', 'California',
'Colorado', 'Connecticut', 'Delaware', 'District of Columbia', 'Florida',
'Georgia', 'Hawaii', 'Idaho', 'Illinois', 'Indiana', 'Iowa', 'Kansas',
'Kentucky', 'Louisiana', 'Maine', 'Maryland', 'Massachusetts', 'Michigan',
'Minnesota', 'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada', 'New
Hampshire', 'New Jersey', 'New Mexico', 'New York', 'North Carolina', 'North
Dakota', 'Ohio', 'Oklahoma', 'Oregon', 'Pennsylvania', 'Rhode Island', 'South
Carolina', 'South Dakota', 'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virginia',
'Washington', 'West Virginia', 'Wisconsin', 'Wyoming'], ['Female', 'Male']],
           codes=[[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9,
10, 10, 11, 11, 12, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19,
20, 20, 21, 21, 22, 22, 23, 23, 24, 24, 25, 25, 26, 26, 27, 27, 28, 28, 29, 29,
30, 30, 31, 31, 32, 32, 33, 33, 34, 34, 35, 35, 36, 36, 37, 37, 38, 38, 39, 39,
40, 40, 41, 41, 42, 42, 43, 43, 44, 44, 45, 45, 46, 46, 47, 47, 48, 48, 49, 49,
50, 50], [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]],
           names=['State', 'Gender'])
```

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()` method. Then, display the `.head()` of the DataFrame.

```
# First, reset the index. Notice the subtle difference; State and Gender are now col
grouped = grouped.reset_index()
grouped.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead tr th {
    text-align: left;
}
```

</style>

	State	Gender	Deaths				
			mean	min	max	std	r
0	Alabama	Female	10753.325000	10	116297	24612.250487	1.078
1	Alabama	Male	10765.850000	10	88930	20813.538537	1.014
2	Alaska	Female	679.975000	13	4727	1154.870455	1.440
3	Alaska	Male	860.357143	12	5185	1411.777392	1.518
4	Arizona	Female	8998.386364	21	133923	26245.941003	1.246

Note how the way 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.

```
grouped.index
```

```
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.

```
# Notice that this causes columns to be MultiIndexed!
grouped.columns
```

```
MultiIndex(levels=[['Deaths', 'Population', 'Gender', 'State'], ['mean', 'min',
'max', 'std', '']],
           codes=[[3, 2, 0, 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` attribute to get the values for the outermost layer of the index.

```
grouped.columns.get_level_values(0)
```

```
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.

```
grouped.columns.get_level_values(1)
```

```
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 a 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 cell below and see if you can understand how it works!

```
# We could also flatten these:
cols0 = grouped.columns.get_level_values(0)
cols1 = grouped.columns.get_level_values(1)
grouped.columns = [col0 + '_' + col1 if col1 != '' else col0 for col0, col1 in list(
# The list comprehension above is more complicated then what we need but creates a r
# 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
```

```
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.

```
grouped.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	State	Gender	Deaths_mean	Deaths_min	Deaths_max	Deaths_

	State	Gender	Deaths_mean	Deaths_min	Deaths_max	Deaths_
0	Alabama	Female	10753.325000	10	116297	24612.25(
1	Alabama	Male	10765.850000	10	88930	20813.53
2	Alaska	Female	679.975000	13	4727	1154.870
3	Alaska	Male	860.357143	12	5185	1411.777
4	Arizona	Female	8998.386364	21	133923	26245.94

Using pivots

Now, we'll gain some practice using the DataFrame'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.

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

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

Gender	Female	Male
State		

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.

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

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

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

Gender	State	Female	Male
4	California	48312.840909	49555.522727

Visualizing Data With Pivot Tables

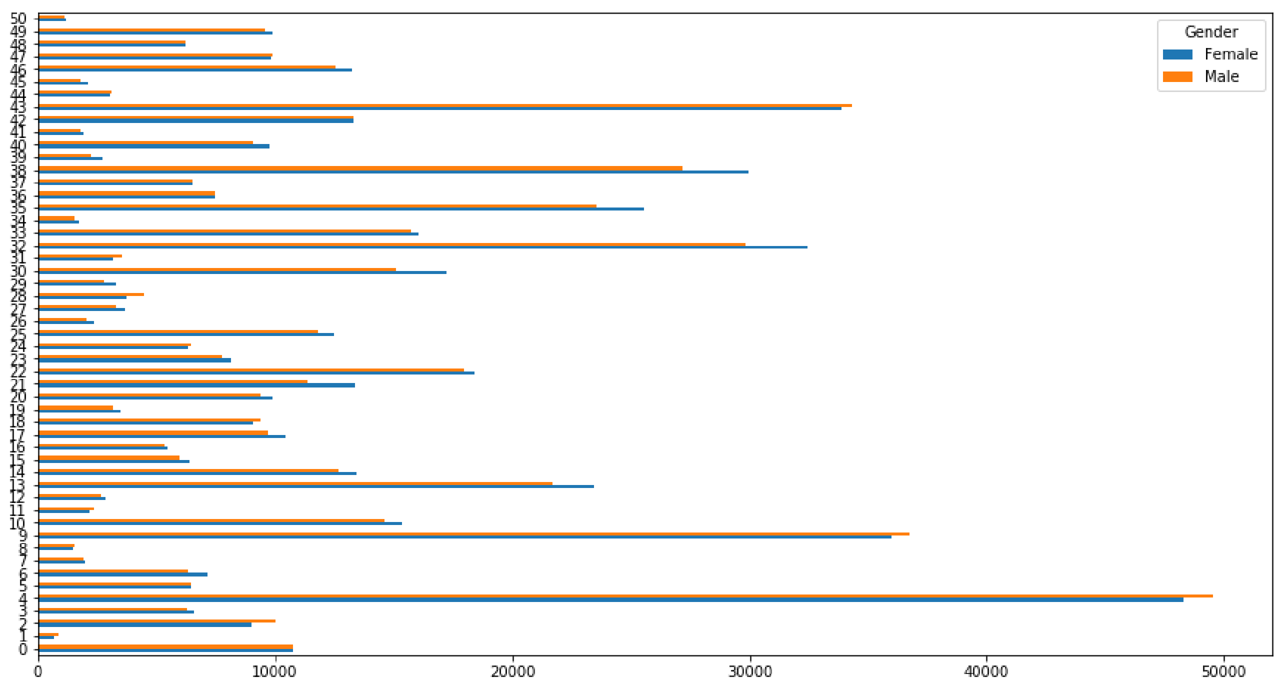
Now, we'll make use of our newly created pivot table to quickly create some visualizations of our data.

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

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

```
# Now let's make a sweet bar chart!!
pivot.plot(kind='barh', figsize=(15,8))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10f6bd4a8>
```



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.

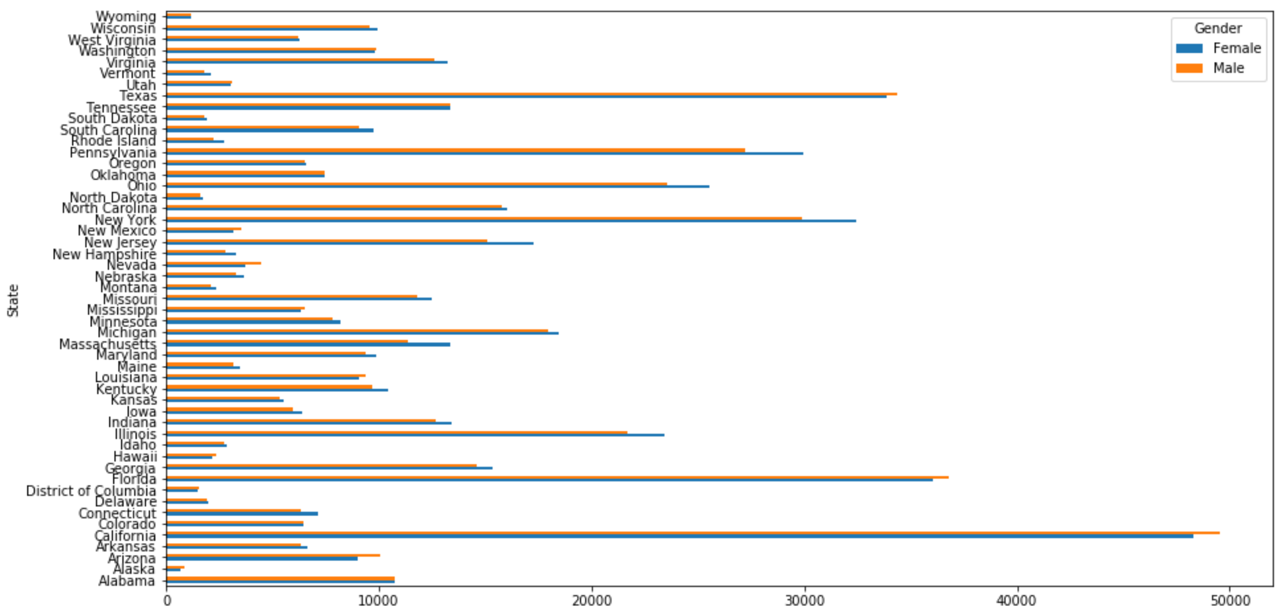
In the cell below:

- 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.

```
# 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 ea
# Let's fix that by making state the index again.
pivot.set_index('State').plot(kind='barh', figsize=(15,8))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1216805f8>
```



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.

```
# 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)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

Gender	State	Female	Male
0	Alabama	10753.325	10765.850000
1	Alaska	679.975	860.357143

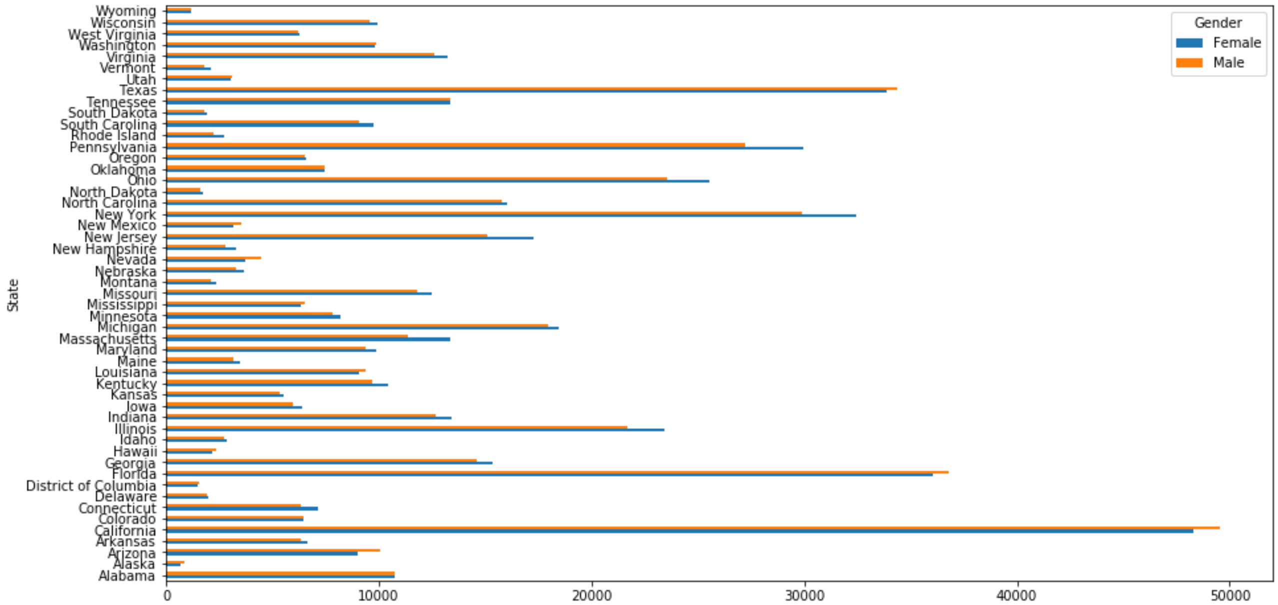
Note that the index has not changed. That's because the code we wrote when we set the index to the `'State'` column returns a copy of the DataFrame object with the index set to `'State'` -- by default, it does not mutate original `pivot` object.

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.

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

```
<matplotlib.axes._subplots.AxesSubplot at 0x121672f28>
```



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

```
pivot.head(2)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

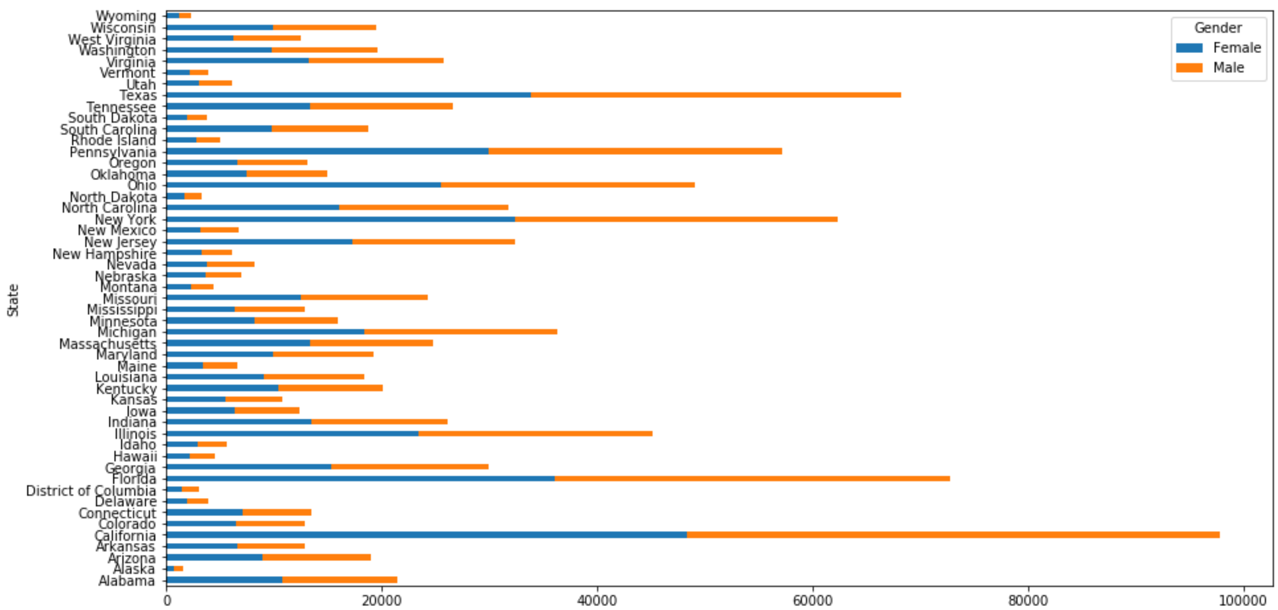
Gender	Female	Male
State		
Alabama	10753.325	10765.850000
Alaska	679.975	860.357143

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.

```
# 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)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x121ccf4a8>
```



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.

```
grouped.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
```

```
    text-align: right;
}
```

</style>

	State	Gender	Deaths_mean	Deaths_min	Deaths_max	Deaths_
0	Alabama	Female	10753.325000	10	116297	24612.25(
1	Alabama	Male	10765.850000	10	88930	20813.53!
2	Alaska	Female	679.975000	13	4727	1154.870·
3	Alaska	Male	860.357143	12	5185	1411.777:
4	Arizona	Female	8998.386364	21	133923	26245.94

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

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

```python
grouped.stack()
```

```
0    State                 Alabama
     Gender                 Female
     Deaths_mean           10753.3
     Deaths_min                 10
     Deaths_max             116297
     Deaths_std            24612.3
     Population_mean   1.07871e+06
     Population_min           2087
     Population_max        4334752
     Population_std    1.40031e+06
1    State                 Alabama
     Gender                   Male
     Deaths_mean           10765.9
     Deaths_min                 10
     Deaths_max              88930
     Deaths_std            20813.5
     Population_mean   1.01495e+06
     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.

```
pivot = pivot.unstack()
pivot
```

```
Gender  State
Female  Alabama                 10753.325000
        Alaska                    679.975000
        Arizona                  8998.386364
        Arkansas                 6621.615385
        California              48312.840909
        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        9078.292683
        South Dakota          1800.500000
        Tennessee            13333.050000
        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 print the `pivot` object to see how things have changed.

```
pivot = pivot.unstack()
pivot
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

State	Alabama	Alaska	Arizona	Arkansas	California
Gender					
Female	10753.325	679.975000	8998.386364	6621.615385	48312.84090
Male	10765.850	860.357143	10036.204545	6301.690476	49555.52272

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!

Summary

In this lab, we learned how to:

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

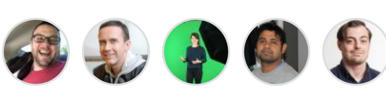
Releases

No releases published

Packages

No packages published

Contributors 5



Languages

- **Jupyter Notebook** 100.0%