Pandas for Panel Data

Thomas J. Sargent and John Stachurski

February 27, 2020

1 Contents

• Overview 2
• Slicing and Reshaping Data 3
• Merging Dataframes and Filling NaNs 4
• Grouping and Summarizing Data 5
• Final Remarks 6
• Exercises 7
• Solutions 8

2 Overview

In an earlier lecture on pandas, we looked at working with simple data sets. Econometricians often need to work with more complex data sets, such as panels. Common tasks include

• Importing data, cleaning it and reshaping it across several axes.
• Selecting a time series or cross-section from a panel.
• Grouping and summarizing data.

pandas (derived from ‘panel’ and ‘data’) contains powerful and easy-to-use tools for solving exactly these kinds of problems.

In what follows, we will use a panel data set of real minimum wages from the OECD to create:

• summary statistics over multiple dimensions of our data
• a time series of the average minimum wage of countries in the dataset
• kernel density estimates of wages by continent

We will begin by reading in our long format panel data from a CSV file and reshaping the resulting DataFrame with pivot_table to build a MultiIndex.

Additional detail will be added to our DataFrame using pandas’ merge function, and data will be summarized with the groupby function.

Most of this lecture was created by Natasha Watkins.
3 Slicing and Reshaping Data

We will read in a dataset from the OECD of real minimum wages in 32 countries and assign it to `realwage`.

The dataset can be accessed with the following link:

In [1]: url1 = 'https://raw.githubusercontent.com/QuantEcon/lecture-source-py/master/source/_static/lecture_specific/pandas_panel/realwage.csv'

In [2]: import pandas as pd

    # Display 6 columns for viewing purposes
    pd.set_option('display.max_columns', 6)

    # Reduce decimal points to 2
    pd.options.display.float_format = '{:,.2f}'.format

    realwage = pd.read_csv(url1)

Let's have a look at what we’ve got to work with

In [3]: realwage.head()  # Show first 5 rows

Out[3]:

<table>
<thead>
<tr>
<th></th>
<th>Unnamed: 0</th>
<th>Time</th>
<th>Country</th>
<th>Pay period</th>
<th>Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2006-01-01</td>
<td>Ireland</td>
<td>Annual</td>
<td>17,132.44</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2007-01-01</td>
<td>Ireland</td>
<td>Annual</td>
<td>18,100.92</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2008-01-01</td>
<td>Ireland</td>
<td>Annual</td>
<td>17,747.41</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2009-01-01</td>
<td>Ireland</td>
<td>Annual</td>
<td>18,580.14</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2010-01-01</td>
<td>Ireland</td>
<td>Annual</td>
<td>18,755.83</td>
</tr>
</tbody>
</table>

The data is currently in long format, which is difficult to analyze when there are several dimensions to the data.

We will use `pivot_table` to create a wide format panel, with a `MultiIndex` to handle higher dimensional data.

`pivot_table` arguments should specify the data (values), the index, and the columns we want in our resulting dataframe.

By passing a list in columns, we can create a `MultiIndex` in our column axis

In [4]: realwage = realwage.pivot_table(values='value',
                                      index='Time',
                                      columns=['Country', 'Series', 'Pay period'])

    realwage.head()
Out[4]:
<table>
<thead>
<tr>
<th>Country</th>
<th>Australia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>In 2015 constant prices at 2015 USD PPPs</td>
</tr>
<tr>
<td>Pay period</td>
<td>Annual Hourly</td>
</tr>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>2006-01-01</td>
<td>20,410.65</td>
</tr>
<tr>
<td>2007-01-01</td>
<td>21,087.57</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>20,718.24</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>20,984.77</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>20,879.33</td>
</tr>
<tr>
<td>Country</td>
<td>United States</td>
</tr>
<tr>
<td>Series</td>
<td>In 2015 constant prices at 2015 USD exchange rates</td>
</tr>
<tr>
<td>Pay period</td>
<td>Annual Hourly</td>
</tr>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>2006-01-01</td>
<td>23,826.64</td>
</tr>
<tr>
<td>2007-01-01</td>
<td>24,616.84</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>24,185.70</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>24,496.84</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>24,373.76</td>
</tr>
</tbody>
</table>

[5 rows x 128 columns]

To more easily filter our time series data, later on, we will convert the index into a Date-TimeIndex.

In [5]: realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)

Out[5]: pandas.core.indexes.datetimes.DatetimeIndex

The columns contain multiple levels of indexing, known as a MultiIndex, with levels being ordered hierarchically (Country > Series > Pay period).

A MultiIndex is the simplest and most flexible way to manage panel data in pandas.

In [6]: type(realwage.columns)
Out[6]: pandas.core.indexes.multi.MultiIndex

In [7]: realwage.columns.names

Out[7]: FrozenList(['Country', 'Series', 'Pay period'])

Like before, we can select the country (the top level of our MultiIndex)

In [8]: realwage['United States'].head()

Out[8]:

<table>
<thead>
<tr>
<th>Time</th>
<th>Pay period</th>
<th>Annual</th>
<th>Hourly</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-01-01</td>
<td>Annual</td>
<td>12,594.40</td>
<td>6.05</td>
</tr>
<tr>
<td>2007-01-01</td>
<td>Annual</td>
<td>12,974.40</td>
<td>6.24</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>Annual</td>
<td>14,097.56</td>
<td>6.78</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>Annual</td>
<td>15,756.42</td>
<td>7.58</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>Annual</td>
<td>16,391.31</td>
<td>7.88</td>
</tr>
</tbody>
</table>

Stacking and unstacking levels of the MultiIndex will be used throughout this lecture to reshape our dataframe into a format we need.

..stack() rotates the lowest level of the column MultiIndex to the row index (.unstack() works in the opposite direction - try it out)

In [9]: realwage.stack().head()
We can also pass in an argument to select the level we would like to stack

```
In [10]: realwage.stack(level='Country').head()
```

```
Out[10]:
```

```
Country Series In 2015 constant prices at 2015 USD PPPs Pay period Time Country
2006-01-01 Annual 21,042.28 10.09 2007-01-01 Annual 21,310.05 10.22 2008-01-01 Annual 21,416.96

Country Series In 2015 constant prices at 2015 USD exchange rates Pay period Time Country
2006-01-01 Annual 20,376.32 9.81 2007-01-01 Annual 20,954.13 10.07 2008-01-01 Annual 20,902.87

Country Series In 2015 constant prices at 2015 USD PPPs Pay period Time Country
2006-01-01 Annual 12,594.40 6.05 2007-01-01 Annual 12,974.40 6.24 2008-01-01 Annual 14,097.56
```

```
[5 rows x 64 columns]
```

We can also pass in an argument to select the level we would like to stack

```
In [10]: realwage.stack(level='Country').head()
```

```
Out[10]: Series In 2015 constant prices at 2015 USD PPPs Pay period Time Country
Annual Hourly
2006-01-01 Australia 20,410.65 10.33 2007-01-01 Annual 21,042.28 10.09 2008-01-01 Annual 21,416.96
Belgium 20,376.32 9.81 2007-01-01 Annual 21,042.28 10.09 2008-01-01 Annual 21,416.96
Brazil 3,310.51 1.41 2007-01-01 Annual 21,042.28 10.09 2008-01-01 Annual 21,416.96
Canada 13,649.69 6.56 2007-01-01 Annual 21,042.28 10.09 2008-01-01 Annual 21,416.96
Chile 5,201.65 2.22 2007-01-01 Annual 21,042.28 10.09 2008-01-01 Annual 21,416.96

Series In 2015 constant prices at 2015 USD exchange rates Pay period Time Country
Annual Hourly
2006-01-01 Australia 23,826.64 12.06 2007-01-01 Annual 20,228.74 9.70 2008-01-01 Annual 21,117.96
Belgium 20,376.32 9.81 2007-01-01 Annual 20,228.74 9.70 2008-01-01 Annual 21,117.96
Brazil 2,032.87 0.87 2007-01-01 Annual 20,228.74 9.70 2008-01-01 Annual 21,117.96
```
Using a `DatetimeIndex` makes it easy to select a particular time period. Selecting one year and stacking the two lower levels of the `MultiIndex` creates a cross-section of our panel data.

```python
In [11]: realwage['2015'].stack(level=(1, 2)).transpose().head()
```

```
Out[11]:

<table>
<thead>
<tr>
<th>Time</th>
<th>2015-01-01</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>In 2015 constant prices at 2015 USD PPPs</td>
<td></td>
</tr>
<tr>
<td>Pay period</td>
<td>Annual Hourly</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Australia</td>
<td>21,715.53 10.99</td>
<td></td>
</tr>
<tr>
<td>Belgium</td>
<td>21,588.12 10.35</td>
<td></td>
</tr>
<tr>
<td>Brazil</td>
<td>4,628.63 2.00</td>
<td></td>
</tr>
<tr>
<td>Canada</td>
<td>16,536.83 7.95</td>
<td></td>
</tr>
<tr>
<td>Chile</td>
<td>6,633.56 2.80</td>
<td></td>
</tr>
</tbody>
</table>
```

For the rest of the lecture, we will work with a dataframe of the hourly real minimum wages across countries and time, measured in 2015 US dollars.

To create our filtered dataframe (`realwage_f`), we can use the `xs` method to select values at lower levels in the multiindex, while keeping the higher levels (countries in this case).

```python
In [12]: realwage_f = realwage.xs(('Hourly', 'In 2015 constant prices at 2015 USD exchange rates'), level=('Pay period', 'Series'), axis=1)
realwage_f.head()
```

```
Out[12]:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>12.06 9.70 0.87 4.21 1.72</td>
<td>12.46 9.82 0.92 4.24 1.71</td>
<td>12.24 9.87 0.96 4.28 1.70</td>
<td>12.40 10.21 1.03 4.27 1.71</td>
<td>12.34 10.05 1.08 4.29 1.68</td>
</tr>
<tr>
<td>Belgium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brazil</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Turkey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>United Kingdom</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>United States</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

[5 rows x 32 columns]
4 Merging Dataframes and Filling NaNs

Similar to relational databases like SQL, pandas has built in methods to merge datasets together.

Using country information from WorldData.info, we will add the continent of each country to `realwage_f` with the `merge` function.

The dataset can be accessed with the following link:

In [13]: url2 = 'https://raw.githubusercontent.com/QuantEcon/lecture-source-py/master/source/_static/lecture_specific/pandas_panel/countries.csv'

In [14]: worlddata = pd.read_csv(url2, sep=';')

worlddata.head()

<table>
<thead>
<tr>
<th>Country (en)</th>
<th>Country (de)</th>
<th>Country (local)</th>
<th>...</th>
<th>Deathrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>Afghanistan</td>
<td>Afganistan/Afqanestan</td>
<td>...</td>
<td>13.70</td>
</tr>
<tr>
<td>Egypt</td>
<td>Ägypten</td>
<td>Misr</td>
<td>...</td>
<td>4.70</td>
</tr>
<tr>
<td>Åland Islands</td>
<td>Alandinseln</td>
<td>Åland</td>
<td>...</td>
<td>0.00</td>
</tr>
<tr>
<td>Albania</td>
<td>Albanien</td>
<td>Shqipëria</td>
<td>...</td>
<td>6.70</td>
</tr>
<tr>
<td>Algeria</td>
<td>Algerien</td>
<td>Al-Jaza’ir/Algérie</td>
<td>...</td>
<td>4.30</td>
</tr>
</tbody>
</table>

We want to merge our new dataframe, `worlddata`, with `realwage_f`.

The pandas `merge` function allows dataframes to be joined together by rows.

Our dataframes will be merged using country names, requiring us to use the transpose of `realwage_f` so that rows correspond to country names in both dataframes.

In [16]: realwage_f.transpose().head()
We can use either left, right, inner, or outer join to merge our datasets:

- left join includes only countries from the left dataset
- right join includes only countries from the right dataset
- outer join includes countries that are in either the left and right datasets
- inner join includes only countries common to both the left and right datasets

By default, `merge` will use an inner join.

Here we will pass `how='left'` to keep all countries in `realwage_f`, but discard countries in `worlddata` that do not have a corresponding data entry `realwage_f`.

This is illustrated by the red shading in the following diagram:

We will also need to specify where the country name is located in each dataframe, which will be the key that is used to merge the dataframes ‘on’.

Our ‘left’ dataframe (`realwage_f.transpose()`) contains countries in the index, so we set `left_index=True`. 

8
Our ‘right’ dataframe (worlddata) contains countries in the ‘Country’ column, so we set right_on='Country'

In [17]: merged = pd.merge(realwage_f.transpose(), worlddata, 
                   how='left', left_index=True, right_on='Country')

merged.head()

Out[17]:
2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00 ...  
  17     12.06  12.46       12.24       ...  
  23     9.70  9.82       9.87       ...  
  32     0.87  0.92       0.96       ...  
 100    6.89  6.96       7.24       ...  
  38     1.42  1.45       1.44       ...  

2016-01-01 00:00:00  Country   Continent
  17     12.98    Australia       Australia
  23      9.76    Belgium        Europe
  32      7.64    Brazil         South America
  100     8.48     Canada       North America
  38      1.91     Chile         South America

[5 rows x 13 columns]

Countries that appeared in realwage_f but not in worlddata will have NaN in the Continent column.

To check whether this has occurred, we can use .isnull() on the continent column and filter the merged dataframe

In [18]: merged[merged['Continent'].isnull()]

Out[18]:
2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00 ...  
  247    3.42  3.74       3.87       ...  
  247    0.23  0.45       0.39       ...  
  247    1.50  1.64       1.71       ...  

2016-01-01 00:00:00  Country    Continent
  247    5.28       Korea        NaN
  247    0.55  Russian Federation  NaN
  247    2.08    Slovak Republic      NaN

[3 rows x 13 columns]

We have three missing values!

One option to deal with NaN values is to create a dictionary containing these countries and their respective continents.

.map() will match countries in merged['Country'] with their continent from the dictionary.

Notice how countries not in our dictionary are mapped with NaN

In [19]: missing_continents = {'Korea': 'Asia',
                            'Russian Federation': 'Europe',
                            'Slovak Republic': 'Europe'}

merged['Country'].map(missing_continents)
We don’t want to overwrite the entire series with this mapping.

`.fillna()` only fills in NaN values in `merged['Continent']` with the mapping, while leaving other values in the column unchanged.

```
In [20]: merged['Continent'] = merged['Continent'].fillna(merged['Country'].map(missing_continents))

# Check for whether continents were correctly mapped
merged[merged['Country'] == 'Korea']
```

```
Out[20]:          2006-01-01  00:00:00  2007-01-01  00:00:00  2008-01-01  00:00:00 ...  
     247       3.42        3.74        3.87        3.97 ...  
     247  2016-01-01  00:00:00  Country  Continent  
     247       5.28        Korea        Asia  
[1 rows x 13 columns]
```

We will also combine the Americas into a single continent - this will make our visualization nicer later on.
To do this, we will use `.replace()` and loop through a list of the continent values we want to replace

```python
In [21]: replace = ['Central America', 'North America', 'South America']
for country in replace:
    merged['Continent'].replace(to_replace=country,
        value='America',
        inplace=True)
```

Now that we have all the data we want in a single DataFrame, we will reshape it back into panel form with a MultiIndex.

We should also ensure to sort the index using `.sort_index()` so that we can efficiently filter our dataframe later on.

By default, levels will be sorted top-down

```python
In [22]: merged = merged.set_index(['Continent', 'Country']).sort_index()
merged.head()
```

```
Out[22]:
Continent Country          ...               ...               ...               ...               ...               ...
America Brazil             0.87     0.92     0.96          ...          1.21          1.21          1.21
                      Canada      6.89     6.96     7.24          ...          8.22          8.35          8.48
                      Chile       1.42     1.45     1.44          ...          1.76          1.81          1.91
                      Colombia     1.01     1.02     1.01          ...          1.13          1.13          1.12
                      Costa Rica nan     nan     nan          ...          2.41          2.56          2.63

[5 rows x 11 columns]
```

While merging, we lost our DatetimeIndex, as we merged columns that were not in datetime format

```python
In [23]: merged.columns
```

```
Out[23]: Index([2006-01-01 00:00:00, 2007-01-01 00:00:00, 2008-01-01 00:00:00, 2009-01-01 00:00:00, 2010-01-01 00:00:00, 2011-01-01 00:00:00, 2012-01-01 00:00:00, 2013-01-01 00:00:00, 2014-01-01 00:00:00, 2015-01-01 00:00:00, 2016-01-01 00:00:00],
dtype='object')
```

Now that we have set the merged columns as the index, we can recreate a DatetimeIndex using `.to_datetime()`
In [24]: merged.columns = pd.to_datetime(merged.columns)
merged.columns = merged.columns.rename('Time')
merged.columns


The DatetimeIndex tends to work more smoothly in the row axis, so we will go ahead and transpose merged

In [25]: merged = merged.transpose()
merged.head()

Out[25]: Continent    America ... Europe
Country   Brazil Canada Chile ... Slovenia Spain United Kingdom
Time 2006-01-01  0.87  6.89  1.42 ...  3.92  3.99  9.81
2007-01-01  0.92  6.96  1.45 ...  3.88  4.10 10.07
2008-01-01  0.96  7.24  1.44 ...  3.96  4.14 10.04
2009-01-01  1.03  7.67  1.52 ...  4.08  4.32 10.15
2010-01-01  1.08  7.94  1.56 ...  4.81  4.30  9.96

[5 rows x 32 columns]

5 Grouping and Summarizing Data

Grouping and summarizing data can be particularly useful for understanding large panel datasets.

A simple way to summarize data is to call an aggregation method on the dataframe, such as .mean() or .max().

For example, we can calculate the average real minimum wage for each country over the period 2006 to 2016 (the default is to aggregate over rows)

In [26]: merged.mean().head(10)

Out[26]: Continent    Country
America     Brazil   1.09
Canada       7.82
Chile        1.62
Colombia     1.07
Costa Rica   2.53
Mexico       0.53
United States 7.15
Asia   Israel   5.95
Japan       6.18
Korea        4.22
dtype: float64

Using this series, we can plot the average real minimum wage over the past decade for each country in our data set
In [27]: import matplotlib.pyplot as plt
   %matplotlib inline
   import matplotlib
   matplotlib.style.use('seaborn')
   
   merged.mean().sort_values(ascending=False).plot(kind='bar',
      title="Average real minimum wage 2006 - 2016")
   
   #Set country labels
   country_labels = 
      merged.mean().sort_values(ascending=False).index.
      get_level_values('Country').tolist()
   plt.xticks(range(0, len(country_labels)), country_labels)
   plt.xlabel('Country')

   plt.show()

Passing in axis=1 to .mean() will aggregate over columns (giving the average minimum wage for all countries over time)

In [28]: merged.mean(axis=1).head()
Out[28]:

<table>
<thead>
<tr>
<th>Time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-01-01</td>
<td>4.69</td>
</tr>
<tr>
<td>2007-01-01</td>
<td>4.84</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>4.90</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>5.08</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>5.11</td>
</tr>
</tbody>
</table>

dtype: float64

We can plot this time series as a line graph.

In [29]:

```python
merged.mean(axis=1).plot()
plt.title('Average real minimum wage 2006 - 2016')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```

We can also specify a level of the MultiIndex (in the column axis) to aggregate over.

In [30]:

```python
merged.mean(level='Continent', axis=1).head()
```

Out[30]:

<table>
<thead>
<tr>
<th>Continent</th>
<th>America</th>
<th>Asia</th>
<th>Australia</th>
<th>Europe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006-01-01</td>
<td>2.80</td>
<td>4.29</td>
<td>10.25</td>
<td>4.80</td>
</tr>
<tr>
<td>2007-01-01</td>
<td>2.85</td>
<td>4.44</td>
<td>10.73</td>
<td>4.94</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>2.99</td>
<td>4.45</td>
<td>10.76</td>
<td>4.99</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>3.23</td>
<td>4.53</td>
<td>10.97</td>
<td>5.16</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>3.34</td>
<td>4.53</td>
<td>10.95</td>
<td>5.17</td>
</tr>
</tbody>
</table>

We can plot the average minimum wages in each continent as a time series.
In [31]: merged.mean(level='Continent', axis=1).plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()

We will drop Australia as a continent for plotting purposes

In [32]: merged = merged.drop('Australia', level='Continent', axis=1)
merged.mean(level='Continent', axis=1).plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
.describe() is useful for quickly retrieving a number of common summary statistics

In [33]: merged.stack().describe()

Out[33]:

<table>
<thead>
<tr>
<th>Continent</th>
<th>America</th>
<th>Asia</th>
<th>Europe</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>69.00</td>
<td>44.00</td>
<td>200.00</td>
</tr>
<tr>
<td>mean</td>
<td>3.19</td>
<td>4.70</td>
<td>5.15</td>
</tr>
<tr>
<td>std</td>
<td>3.02</td>
<td>1.56</td>
<td>3.82</td>
</tr>
<tr>
<td>min</td>
<td>0.52</td>
<td>2.22</td>
<td>0.23</td>
</tr>
<tr>
<td>25%</td>
<td>1.03</td>
<td>3.37</td>
<td>2.02</td>
</tr>
<tr>
<td>50%</td>
<td>1.44</td>
<td>5.48</td>
<td>3.54</td>
</tr>
<tr>
<td>75%</td>
<td>6.96</td>
<td>5.95</td>
<td>9.70</td>
</tr>
<tr>
<td>max</td>
<td>8.48</td>
<td>6.65</td>
<td>12.39</td>
</tr>
</tbody>
</table>

This is a simplified way to use groupby.

Using groupby generally follows a ‘split-apply-combine’ process:

- split: data is grouped based on one or more keys
- apply: a function is called on each group independently
- combine: the results of the function calls are combined into a new data structure

The groupby method achieves the first step of this process, creating a new DataFrameGroupBy object with data split into groups.

Let’s split merged by continent again, this time using the groupby function, and name the resulting object grouped

In [34]: grouped = merged.groupby(level='Continent', axis=1)

grouped
Calling an aggregation method on the object applies the function to each group, the results of which are combined in a new data structure.

For example, we can return the number of countries in our dataset for each continent using `.size()`.

In this case, our new data structure is a `Series`.

```
In [35]: grouped.size()
```

```
Out[35]:

<table>
<thead>
<tr>
<th>Continent</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>America</td>
<td>7</td>
</tr>
<tr>
<td>Asia</td>
<td>4</td>
</tr>
<tr>
<td>Europe</td>
<td>19</td>
</tr>
</tbody>
</table>

```

Calling `.get_group()` to return just the countries in a single group, we can create a kernel density estimate of the distribution of real minimum wages in 2016 for each continent.

```
grouped.groups.keys() will return the keys from the groupby object.

In [36]: import seaborn as sns

continents = grouped.groups.keys()

for continent in continents:
    sns.kdeplot(grouped.get_group(continent)["2015"].unstack(),
                label=continent,
                shade=True)
```

```
plt.title('Real minimum wages in 2015')
plt.xlabel('US dollars')
plt.show()
```
6 Final Remarks

This lecture has provided an introduction to some of pandas’ more advanced features, including multiindices, merging, grouping and plotting.

Other tools that may be useful in panel data analysis include xarray, a python package that extends pandas to N-dimensional data structures.

7 Exercises

7.1 Exercise 1

In these exercises, you’ll work with a dataset of employment rates in Europe by age and sex from Eurostat.

The dataset can be accessed with the following link:

In [37]: url3 = 'https://raw.githubusercontent.com/QuantEcon/lecture-source-py/master/source/_static/lecture_specific/pandas_panel/employ.csv'

Reading in the CSV file returns a panel dataset in long format. Use .pivot_table() to construct a wide format dataframe with a MultiIndex in the columns.

Start off by exploring the dataframe and the variables available in the MultiIndex levels.
Write a program that quickly returns all values in the MultiIndex.

7.2 Exercise 2

Filter the above dataframe to only include employment as a percentage of ‘active population’.
Create a grouped boxplot using seaborn of employment rates in 2015 by age group and sex.

Hint: GEO includes both areas and countries.

8 Solutions

8.1 Exercise 1

In [38]: employ = pd.read_csv(url3)
employ = employ.pivot_table(values='Value',
                           index=['DATE'],
                           columns=['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])
employ.index = pd.to_datetime(employ.index) # ensure that dates are
in datetime format
employ.head()
This is a large dataset so it is useful to explore the levels and variables available

```
In [39]: employ.columns.names

Out[39]: FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])

Variables within levels can be quickly retrieved with a loop

```

```
In [40]: for name in employ.columns.names:
   ...:     print(name, employ.columns.get_level_values(name).unique())

UNIT Index(['Percentage of total population', 'Thousand persons'],
           dtype='object',
           name='UNIT')
AGE Index(['From 15 to 24 years', 'From 25 to 54 years', 'From 55 to 64 years'],
          dtype='object',
          name='AGE')
SEX Index(['Females', 'Males', 'Total'], dtype='object', name='SEX')
```
8.2 Exercise 2

To easily filter by country, swap GEO to the top level and sort the MultiIndex

```
In [41]: employ.columns = employ.columns.swaplevel(0,-1)
   employ = employ.sort_index(axis=1)
```

We need to get rid of a few items in GEO which are not countries.

A fast way to get rid of the EU areas is to use a list comprehension to find the level values in GEO that begin with ‘Euro’

```
In [42]: geo_list = employ.columns.get_level_values('GEO').unique().tolist()
countries = [x for x in geo_list if not x.startswith('Euro')]
   employ = employ[countries]
   employ.columns.get_level_values('GEO').unique()
```

```
Out[42]: Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech Republic',
               'Denmark', 'Estonia', 'Finland',
               'Former Yugoslav Republic of Macedonia, the', 'France',
               'France (metropolitan)',
               'Germany (until 1990 former territory of the FRG)', 'Greece',
               'Hungary',
               'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
               'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
               'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
               'United Kingdom'],
   dtype='object', name='GEO')
```

Select only percentage employed in the active population from the dataframe

```
In [43]: employ_f = employ.xs((['Percentage of total population', 'Active population'],
                           level=('UNIT', 'INDIC_EM'),
                          axis=1)
   employ_f.head()
```
Out[43]:

<table>
<thead>
<tr>
<th>GEO</th>
<th>Austria</th>
<th>...</th>
<th>United Kingdom</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGE</td>
<td>From 15 to 24 years</td>
<td>...</td>
<td>From 55 to 64 years</td>
</tr>
<tr>
<td>SEX</td>
<td>Females Males Total</td>
<td>...</td>
<td>Females Males</td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>2007-01-01</td>
<td>56.00 62.90 59.40</td>
<td>...</td>
<td>49.90 68.90</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>56.20 62.90 59.50</td>
<td>...</td>
<td>50.20 69.80</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>56.20 62.90 59.50</td>
<td>...</td>
<td>50.60 70.30</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>54.00 62.60 58.30</td>
<td>...</td>
<td>51.10 69.20</td>
</tr>
<tr>
<td>2011-01-01</td>
<td>54.80 63.60 59.20</td>
<td>...</td>
<td>51.30 68.40</td>
</tr>
</tbody>
</table>

[5 rows x 306 columns]

Drop the ‘Total’ value before creating the grouped boxplot

In [44]: employ_f = employ_f.drop('Total', level='SEX', axis=1)

In [45]: box = employ_f['2015'].unstack().reset_index()
sns.boxplot(x="AGE", y=0, hue="SEX", data=box, palette="husl", showfliers=False)
plt.xlabel('')
plt.xticks(rotation=35)
plt.ylabel('Percentage of population (%)')
plt.title('Employment in Europe (2015)')
plt.legend(bbox_to_anchor=(1,0.5))
plt.show()