Recap

Previous lecture: basics of pandas
Series and DataFrames
Indexing, changing entries
Function application

This lecture: more complicated operations
Statistical computations
Group-By operations
Reshaping, stacking and pivoting
Recap

Previous lecture: basics of pandas
  Series and DataFrames
  Indexing, changing entries
  Function application

This lecture: more complicated operations
  Statistical computations
  Group-By operations
  Reshaping, stacking and pivoting

Caveat: pandas is a large, complicated package, so I will not endeavor to mention every feature here. These slides should be enough to get you started, but there's no substitute for reading the documentation.
pct_change method is supported by both Series and DataFrames. Series.pct_change returns a new Series representing the step-wise percent change. 

pct_change includes control over how missing data is imputed, how large a time-lag to use, etc. See documentation for more detail: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.pct_change.html
Percent change over time

```
pct_change operates on columns of a DataFrame, by
default. Periods argument specifies the time-lag to use
in computing percent change. So periods=2 looks at
percent change compared to two time steps ago.
```

**Note:** pandas has extensive support for time series
data, which we mostly won’t talk about in this course.

```
pct_change includes control over how missing
data is imputed, how large a time-lag to use, etc.
See documentation for more detail:
```

```
1 df.pct_change(periods=2)
```

```
0  NaN   NaN   NaN   NaN
1   NaN   NaN   NaN   NaN
2 -0.720087 1.535504 -1.857284   3.743931
3 -1.047838 -0.737821   0.779726  -4.477898
4  5.579538 -2.298878  -2.486674  -0.451508
5 -0.390876  1.331029  -0.696448 -1.017590
```
Computing covariances

**cov** method computes covariance between a Series and another Series.

```python
1 s1 = pd.Series(np.random.randn(1000))
2 s2 = pd.Series(0.1*s1+np.random.randn(1000))
3 sl.cov(s2)
```

0.1522727637202401

**cov** method is also supported by DataFrame, but instead computes a new DataFrame of covariances between columns.

```python
1 df.cov()
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.305249</td>
<td>-0.364416</td>
<td>0.815636</td>
<td>0.189141</td>
</tr>
<tr>
<td>1</td>
<td>2.425535</td>
<td>-1.082098</td>
<td>-0.771105</td>
<td>0.363440</td>
</tr>
<tr>
<td>2</td>
<td>-0.085443</td>
<td>-0.923977</td>
<td>-0.699232</td>
<td>0.897274</td>
</tr>
<tr>
<td>3</td>
<td>-0.116032</td>
<td>-0.283703</td>
<td>-1.372355</td>
<td>-1.264006</td>
</tr>
<tr>
<td>4</td>
<td>-0.562175</td>
<td>1.200134</td>
<td>1.039529</td>
<td>0.492148</td>
</tr>
<tr>
<td>5</td>
<td>-0.070678</td>
<td>-0.661320</td>
<td>-0.416581</td>
<td>0.022234</td>
</tr>
</tbody>
</table>

Pairwise correlations

DataFrame `corr` method computes correlations between columns (use `axis` keyword to change this behavior). The `method` argument controls which correlation score to use (default is Pearson’s correlation).

```
1 df = pd.DataFrame(np.random.randn(1000, 5),
2     columns=['a', 'b', 'c', 'd', 'e'])
3 df.corr(method='spearman')
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.00</td>
<td>0.01</td>
<td>-0.03</td>
<td>0.00</td>
<td>-0.49</td>
</tr>
<tr>
<td>b</td>
<td>0.01</td>
<td>1.00</td>
<td>-0.00</td>
<td>0.00</td>
<td>-0.18</td>
</tr>
<tr>
<td>c</td>
<td>-0.03</td>
<td>-0.00</td>
<td>1.00</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>d</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>1.00</td>
<td>0.05</td>
</tr>
<tr>
<td>e</td>
<td>-0.49</td>
<td>-0.18</td>
<td>0.03</td>
<td>0.05</td>
<td>1.00</td>
</tr>
</tbody>
</table>

```
1 df.corr(method='kendall')
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.00</td>
<td>0.01</td>
<td>-0.02</td>
<td>0.00</td>
<td>-0.03</td>
</tr>
<tr>
<td>b</td>
<td>0.01</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-0.12</td>
</tr>
<tr>
<td>c</td>
<td>-0.02</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>d</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.03</td>
</tr>
<tr>
<td>e</td>
<td>-0.03</td>
<td>-0.12</td>
<td>0.02</td>
<td>0.03</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Ranking data

The `rank` method returns a new Series whose values are the data ranks. Ties are broken by assigning the mean rank to both values.
Ranking data

By default, `rank` ranks columns of a DataFrame individually.

```python
df.rank()
```

Rank rows instead by supplying an `axis` argument.

```python
df.rank(1)
```

**Note:** more complicated ranking of whole rows (i.e., sorting whole rows rather than sorting columns individually) is possible, but requires we define an ordering on Series.
Group By: reorganizing data

“Group By” operations are a concept from databases
  Splitting data based on some criteria
  Applying functions to different splits
  Combining results into a single data structure

Fundamental object: pandas GroupBy objects
Group By: reorganizing data

```python
1 df = pd.DataFrame({'A': ['plant', 'animal', 'plant', 'plant'],
                   'B': ['apple', 'goat', 'kiwi', 'grape'],
                   'C': np.random.randn(4),
                   'D': np.random.randn(4))

1 df.groupby('A')
```

DataFrame `groupby` method returns a pandas `groupby` object.
Group By: reorganizing data

Every `groupby` object has an attribute `groups`, which is a dictionary with maps group labels to the indices in the DataFrame.

In this example, we are splitting on the column ‘A’, which has two values: ‘plant’ and ‘animal’, so the groups dictionary has two keys.
Group By: reorganizing data

Every `groupby` object has an attribute `groups`, which is a dictionary with maps group labels to the indices in the DataFrame.

In this example, we are splitting on the column `A`, which has two values: `plant` and `animal`, so the groups dictionary has two keys.
Group By: aggregation

Split on group ‘A’, then compute the means within each group. Note that columns for which means are not supported are removed, so column ‘B’ doesn’t show up in the result.
Here we’re building a hierarchically-indexed Series (i.e., multi-indexed), recording (fictional) scores of students by major and handedness.

Suppose I want to collapse over handedness to get average scores by major. In essence, I want to group by major and ignore handedness.
Suppose I want to collapse over handedness to get average scores by major. In essence, I want to group by major and ignore handedness.

Group by the 0-th level of the hierarchy (i.e., ‘major’), and take means.

We could have equivalently written `groupby('major'), here.`
Group By: examining groups

groupby.get_group lets us pick out an individual group. Here, we're grabbing just the data from the ‘econ’ group, after grouping by ‘major’.
Group By: aggregation

Similar aggregation to what we did a few slides ago, but now we have a DataFrame instead of a Series.
Group By: aggregation

Similar aggregation to what we did a few slides ago, but now we have a DataFrame instead of a Series.

Groupby objects also support the `aggregate` method, which is often more convenient.

```python
g = df.groupby('handedness')
g.aggregate(np.sum)
```

```
df.groupby('handedness').mean()
```

```
<table>
<thead>
<tr>
<th>handedness</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>7</td>
<td>-2.184352</td>
</tr>
<tr>
<td>right</td>
<td>7</td>
<td>-1.034337</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>handedness</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>1.75</td>
<td>-0.546088</td>
</tr>
<tr>
<td>right</td>
<td>1.75</td>
<td>-0.258584</td>
</tr>
</tbody>
</table>
```
Transforming data

From the documentation: “The transform method returns an object that is indexed the same (same size) as the one being grouped.”

Building a time series, indexed by year-month-day.

Suppose we want to standardize these scores within each year.

Group the data according to the output of the key function, apply the given transformation within each group, then un-group the data.

Important point: the result of `groupby.transform` has the same dimension as the original DataFrame or Series.
Filtering data

From the documentation: “The argument of filter must be a function that, applied to the group as a whole, returns True or False.”

So this will throw out all the groups with sum <= 2.

Like `transform`, the result is ungrouped.
Combining DataFrames

The `pandas.concat` function concatenates DataFrames into a single DataFrame. Repeated indices remain repeated in the resulting DataFrame. Missing values get NaN.

`pandas.concat` accepts numerous optional arguments for finer control over how concatenation is performed. See the documentation for more.
Merges and joins

Pandas DataFrames support many common database operations. Most notably, join and merge operations.

We’ll learn about these when we discuss SQL later in the semester. So we won’t discuss them here.

Important: What we learn for SQL later has analogues in pandas.

If you are already familiar with SQL, you might like to read this:
Pivoting and Stacking

Data in this format is usually called **stacked**. It is common to store data in this form in a file, but once it’s read into a table, it often makes more sense to create columns for A, B and C. That is, we want to **unstack** this DataFrame.
The `pivot` method takes care of unstacking DataFrames. We supply indices for the new DataFrame, and tell it to turn the variable column in the old DataFrame into a set of column names in the unstacked one.

https://en.wikipedia.org/wiki/Pivot_table
Pivoting and Stacking

How do we stack this? That is, how do we get a non-pivot version of this DataFrame? The answer is to use the DataFrame `stack` method.

```python
import pandas as pd
import numpy as np

tuples = list(zip(*[['bird','bird','goat','goat'],
                    ['x', 'y', 'x', 'y']])))
index = pd.MultiIndex.from_tuples(tuples, names=['animal', 'cond'])
df = pd.DataFrame(np.random.randn(4, 2),
                  index=index, columns=['A', 'B'])
df
```

<table>
<thead>
<tr>
<th>animal</th>
<th>cond</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bird</td>
<td>x</td>
<td>0.699732</td>
<td>-1.407296</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>0.810211</td>
<td>1.249299</td>
</tr>
<tr>
<td>goat</td>
<td>x</td>
<td>-0.909280</td>
<td>0.184450</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>-0.755891</td>
<td>-0.957222</td>
</tr>
</tbody>
</table>
Pivoting and Stacking

The DataFrame `stack` method makes a stacked version of the calling DataFrame. In the event that the resulting column index set is trivial, the result is a Series. Note that `df.stack()` no longer has columns A or B. The column labels A and B have become an extra index.
Here is a more complicated example. Notice that the column labels have a three-level hierarchical structure.

There are multiple ways to stack this data. At one extreme, we could make all three levels into columns. At the other extreme, we could choose only one to make into a column.
### Pivoting and Stacking

Stack only according to level 1 (i.e., the animal column index).

Missing animal x cond x hair_length conditions default to NaN.
Pivoting and Stacking

Stacking across all three levels yields a Series, since there is no longer any column structure. This is often called **flattening** a table.

Notice that the NaN entries are not necessary here, since we have an entry in the Series only for entries of the original DataFrame.
Plotting DataFrames

cumsum gets partial sums, just like in numpy.

Note: this requires that you have imported matplotlib.

Note that legend is automatically populated and x-ticks are automatically date formatted.