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: [https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.pct_change.html](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.pct_change.html)
Computing covariances

cov method computes covariance between a Series and another Series.

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

cov supports extra arguments for further specifying behavior:
Pairwise correlations

DataFrame `corr` method computes correlations between columns (use `axis` keyword to change this behavior). `method` argument controls which correlation score to use (default is Pearson's correlation).
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.
By default, `rank` ranks columns of a DataFrame individually.

Rank rows instead by supplying an `axis` argument.

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
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.

The important point is that the `groupby` object is storing information about how to partition the rows of the original DataFrame according to the argument(s) passed to the `groupby` method.

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>plant</td>
<td>apple</td>
<td>0.529326</td>
<td>-0.796997</td>
</tr>
<tr>
<td>1</td>
<td>animal</td>
<td>goat</td>
<td>-0.901377</td>
<td>-0.670747</td>
</tr>
<tr>
<td>2</td>
<td>plant</td>
<td>kiwi</td>
<td>1.203032</td>
<td>1.162924</td>
</tr>
<tr>
<td>3</td>
<td>plant</td>
<td>grape</td>
<td>-0.740208</td>
<td>1.184488</td>
</tr>
</tbody>
</table>

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.

```python
df.groupby('A').mean()
```

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>animal</td>
<td>-0.901377</td>
<td>-0.670747</td>
</tr>
<tr>
<td>plant</td>
<td>0.330717</td>
<td>0.516805</td>
</tr>
</tbody>
</table>
Group By: aggregation

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

```
+---------+---------+        +---------+---------+
| major   | handedness |          | major   | handedness |
|---------+-----------+        |---------+-----------|
| math    | left      | 1 -0.856890 | econ    | left      | 1 -0.707796 |
|         | right     | 1 0.425160  |         | right     | 1 -1.944487 |
|         |           |            |         |           |             |
|         | left      | 2 0.341265  |         | right     | 2 -0.938632 |
|         | right     | 2 -0.960931 |         |           |             |
|         |           |            |         | left      | 3 1.423622  |
|         | right     | 3 -1.034337 |         |           |             |
```

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

```
+---------+---------+        +---------+---------+
| handedness |         |          | handedness |         |
|-----------|---------+        |-----------+---------+
| left      | 7 -2.184352 |        | left      | 1.75 -0.546088 |
| right     | 7 -1.034337 |        | right     | 1.75 -0.258584 |
```
### Transforming data

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

Suppose we want to standardize these scores within each year.

```python
index = pd.date_range('10/1/1999', periods=1100)
ts = pd.Series(np.random.normal(0.5, 2, 1100), index)
ts.head()
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999-10-01</td>
<td>-1.283451</td>
</tr>
<tr>
<td>1999-10-02</td>
<td>0.468645</td>
</tr>
<tr>
<td>1999-10-03</td>
<td>2.796156</td>
</tr>
<tr>
<td>1999-10-04</td>
<td>0.449197</td>
</tr>
<tr>
<td>1999-10-05</td>
<td>1.647331</td>
</tr>
</tbody>
</table>

Freq: D, dtype: float64

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

```python
key = lambda d: d.year
zscore = lambda x: (x - x.mean()) / x.std()
transformed = ts.groupby(key).transform(zscore)
transformed.head()
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Transformed Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999-10-01</td>
<td>-1.097395</td>
</tr>
<tr>
<td>1999-10-02</td>
<td>-0.243334</td>
</tr>
<tr>
<td>1999-10-03</td>
<td>0.891214</td>
</tr>
<tr>
<td>1999-10-04</td>
<td>-0.252814</td>
</tr>
<tr>
<td>1999-10-05</td>
<td>0.331218</td>
</tr>
</tbody>
</table>

Freq: D, dtype: float64

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

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

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

**pandas.concat** function concatenates DataFrames into a single DataFrame.

Repeated indices remain repeated in the resulting DataFrame.

**pandas.concat** accepts numerous optional arguments for finer control over how concatenation is performed. See the documentation for more.

Missing values get NaN.
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.
Pivoting and Stacking

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
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>
The DataFrame \texttt{stack} method makes a stacked version of the calling DataFrame. In the event that the resulting column index set is a trivial, the result is a Series. Note that \texttt{df.stack()} no longer has columns A or B. The column labels A and B have become an extra index.
Pivoting and Stacking

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.

```
1 df.stack(level=1)
```
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.
Plotting DataFrames

DataFrames.plot() method is largely identical to matplotlib.pyplot
So you already mostly know how to use it!

Additional plot types:
https://pandas.pydata.org/pandas-docs/stable/visualization.html#other-plots

More advanced plotting tools:
https://pandas.pydata.org/pandas-docs/stable/visualization.html#plotting-tools
Readings

Required:
  Group By:
  Reshaping and pivoting:

Recommended:
  Merge, join and concatenation:
  Time series functionality: