STATS 701
Data Analysis using Python

Lecture 14: Advanced `pandas`

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.

Percent change over time

`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

```python
s = pd.Series(np.random.randn(8))
s
```

```
0   -0.669520
1   -0.864352
2   -1.686718
3    0.014609
4   -2.199920
5   -0.505137
6   -0.403893
7   -0.358685
dtype: float64
```

```python
s.pct_change()
```

```
0           NaN
1      0.291003
2      0.951425
3     -1.008661
4   -151.589298
5     -0.770384
6     -0.200428
7     -0.111931
dtype: float64
```

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

	0	1	2	3
0	-0.305249	-0.364416	0.815636	0.189141
1	2.425535	-1.082098	-0.771105	0.363440
2	-0.085443	-0.923977	-0.699232	0.897274
3	-0.116032	-0.283703	-1.372355	-1.264006
4	-0.562175	1.200134	1.039529	0.492148
5	-0.070678	-0.661320	-0.416581	0.022234

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

	0	1	2	3
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.

```
s1 = pd.Series(np.random.randn(1000))
s2 = pd.Series(0.1*s1+np.random.randn(1000))
s1.cov(s2)
```

```
0.1522727637202401
```

	0	1	2	3
0	-0.305249	-0.364416	0.815636	0.189141
1	2.425535	-1.082098	-0.771105	0.363440
2	-0.085443	-0.923977	-0.699232	0.897274
3	-0.116032	-0.283703	-1.372355	-1.264006
4	-0.562175	1.200134	1.039529	0.492148
5	-0.070678	-0.661320	-0.416581	0.022234

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

```
df.cov()
```

	0	1	2	3
0	1.208517	-0.515225	-0.430870	0.093096
1	-0.515225	0.673964	0.520126	-0.021969
2	-0.430870	0.520126	0.911544	0.329498
3	0.093096	-0.021969	0.329498	0.546332

`cov` supports extra arguments for further specifying behavior:
https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.cov.html

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.

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

	a	b	c	d	e
a	1.000000	0.018325	-0.029441	0.002467	-0.048051
b	0.018325	1.000000	-0.000091	0.004212	-0.018435
c	-0.029441	-0.000091	1.000000	0.016103	0.034150
d	0.002467	0.004212	0.016103	1.000000	0.053519
e	-0.048051	-0.018435	0.034150	0.053519	1.000000

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

	a	b	c	d	e
a	1.000000	0.012264	-0.019075	0.001333	-0.032745
b	0.012264	1.000000	0.000212	0.002515	-0.012168
c	-0.019075	0.000212	1.000000	0.009630	0.022326
d	0.001333	0.002515	0.009630	1.000000	0.035872
e	-0.032745	-0.012168	0.022326	0.035872	1.000000

Ranking data

`rank` method returns a new Series whose values are the data ranks.

Ties are broken by assigning the mean rank to both values.

```
s = pd.Series(np.random.np.random.randn(5),
              index=list('abcde'))
s
```

```
a    1.804688
b   -1.203916
c    1.055365
d   -0.048237
e    1.659330
dtype: float64
```

```
s.rank()
```

```
a    5.0
b    1.0
c    3.0
d    2.0
e    4.0
dtype: float64
```

```
s[0] = s[1] = 0
s.rank()
```

```
a    2.5
b    2.5
c    4.0
d    1.0
e    5.0
dtype: float64
```

Ranking data

	0	1	2	3	4
0	-0.606576	-0.892385	0.891247	-0.280582	0.601239
1	-1.036933	0.905388	0.012123	-2.497602	0.501482
2	0.387677	0.850437	-1.578854	-0.263305	0.540390
3	-0.631557	-0.528819	0.561295	0.955113	0.980433

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

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

	0	1	2	3	4
0	3.0	1.0	4.0	2.0	3.0
1	1.0	4.0	2.0	1.0	1.0
2	4.0	3.0	1.0	3.0	2.0
3	2.0	2.0	3.0	4.0	4.0

Rank rows instead by supplying an `axis` argument.

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

	0	1	2	3	4
0	2.0	1.0	5.0	3.0	4.0
1	2.0	5.0	3.0	1.0	4.0
2	3.0	5.0	1.0	2.0	4.0
3	1.0	2.0	3.0	4.0	5.0

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

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

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

DataFrame `groupby` method returns a `pandas groupby` object.

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

```
<pandas.core.groupby.DataFrameGroupBy object at 0x11fe88bd0>
```

Group By: reorganizing data

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

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

```
<pandas.core.groupby.DataFrameGroupBy object at 0x11fe88bd0>
```

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

```
{'animal': Int64Index([1], dtype='int64'),
 'plant': Int64Index([0, 2, 3], dtype='int64')}
```

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

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

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

```
<pandas.core.groupby.DataFrameGroupBy
```

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

```
{'animal': Int64Index([1], dtype='int64'),
 'plant': Int64Index([0, 2, 3], dtype='int64')}
```

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

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

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

A	C	D
animal	-0.901377	-0.670747
plant	0.330717	0.516805

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.

Group By: aggregation

```
arrs = [['math', 'math', 'econ', 'econ', 'stats', 'stats'],
            ['left', 'right', 'left', 'right', 'left', 'right']]
index = pd.MultiIndex.from_arrays(arrs, names=['major', 'handedness'])
s = pd.Series(np.random.randn(6), index=index)
s
```

```
major  handedness
math   left         -2.015677
       right         0.537438
econ   left          1.071951
       right        -0.504158
stats  left          1.204159
       right        -0.288676
dtype: float64
```

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.

Group By: aggregation

```
major  handedness
math   left         -2.015677
       right         0.537438
econ   left          1.071951
       right        -0.504158
stats  left          1.204159
       right        -0.288676
dtype: float64
```

```
s.groupby(level=0).mean()
```

```
major
econ      0.283897
math     -0.739120
stats     0.457741
dtype: float64
```

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

```
s
```

```
major  handedness
math   left         -2.015677
       right         0.537438
econ   left          1.071951
       right        -0.504158
stats  left          1.204159
       right        -0.288676
dtype: float64
```

```
s.groupby('major').get_group('econ')
```

```
major  handedness
econ   left          1.071951
       right        -0.504158
dtype: float64
```

`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.

major	handedness	A	B
math	left	1	-0.856890
	right	1	0.425160
econ	left	1	-0.707796
	right	1	-1.944487
stats	left	2	0.341265
	right	2	-0.938632
phys	left	3	-0.960931
	right	3	1.423622

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

handedness	A	B
left	1.75	-0.546088
right	1.75	-0.258584

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.

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

handedness	A	B
left	7	-2.184352
right	7	-1.034337

major	handedness	A	B
math	left	1	-0.856890
	right	1	0.425160
econ	left	1	-0.707796
	right	1	-1.944487
stats	left	2	0.341265
	right	2	-0.938632
phys	left	3	-0.960931
	right	3	1.423622

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

handedness	A	B
left	1.75	-0.546088
right	1.75	-0.258584

Transforming data

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

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

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

```
1999-10-01   -1.283451
1999-10-02    0.468645
1999-10-03    2.796156
1999-10-04    0.449197
1999-10-05    1.647331
Freq: D, dtype: float64
```

Suppose we want to standardize these scores within each year.

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

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

```
1999-10-01   -1.097395
1999-10-02   -0.243334
1999-10-03    0.891214
1999-10-04   -0.252814
1999-10-05    0.331218
Freq: D, dtype: float64
```

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

Filtering data

```
sf = pd.Series([1, 1, 2, 2, 3, 3])
sf
```

```
0    1
1    1
2    2
3    2
4    3
5    3
dtype: int64
```

```
sf.groupby(sf).filter(lambda x: x.sum() > 2)
```

```
2    2
3    2
4    3
5    3
dtype: int64
```

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

```
df1 = pd.DataFrame({'A':np.random.randn(4),
                    'B':np.random.randn(4),
                    'C':np.random.randn(4)},
                   index=[0,1,2,3])
df2 = pd.DataFrame({'A':np.random.randn(4),
                    'B':np.random.randn(4)},
                   index=[3,4,5,6])
pd.concat([df1,df2])
```

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

	A	B	C
0	0.755669	1.497149	0.889586
1	-0.197404	0.674905	1.131785
2	0.341409	0.632993	0.495411
3	0.646052	-0.809168	-0.708263
3	0.508306	-0.070561	NaN
4	1.172885	-0.518003	NaN
5	-0.103887	-0.479715	NaN
6	0.596387	-2.156612	NaN

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:
https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html

Pivoting and Stacking

	date	variable	value
0	2000-01-03	A	1.234594
1	2000-01-04	A	0.661894
2	2000-01-05	A	0.810323
3	2000-01-03	B	-0.156366
4	2000-01-04	B	0.798020
5	2000-01-05	B	-0.360506
6	2000-01-03	C	0.375464
7	2000-01-04	C	0.413346
8	2000-01-05	C	-0.071480
9	2000-01-03	D	0.108641
10	2000-01-04	D	-0.738962
11	2000-01-05	D	0.460154

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

	date	variable	value
0	2000-01-03	A	1.234594
1	2000-01-04	A	0.661894
2	2000-01-05	A	0.810323
3	2000-01-03	B	-0.156366
4	2000-01-04	B	0.798020
5	2000-01-05	B	-0.360506
6	2000-01-03	C	0.375464
7	2000-01-04	C	0.413346
8	2000-01-05	C	-0.071480
9	2000-01-03	D	0.108641
10	2000-01-04	D	-0.738962
11	2000-01-05	D	0.460154

https://en.wikipedia.org/wiki/Pivot_table

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.

```python
df.pivot(index='date',
         columns='variable',
         values='value')
```

variable	A	B	C	D
date				
2000-01-03	1.234594	-0.156366	0.375464	0.108641
2000-01-04	0.661894	0.798020	0.413346	-0.738962
2000-01-05	0.810323	-0.360506	-0.071480	0.460154

Pivoting and Stacking

```
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
```

		A	B
animal	cond		
bird	x	0.699732	-1.407296
	y	0.810211	1.249299
goat	x	-0.909280	0.184450
	y	-0.755891	-0.957222

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.

Pivoting and Stacking

animal	cond	A	B
bird	x	0.699732	-1.407296
	y	0.810211	1.249299
goat	x	-0.909280	0.184450
	y	-0.755891	-0.957222

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.

```
df.stack()
```

```
animal  cond
bird    x     A    0.699732
              B   -1.407296
        y     A    0.810211
              B    1.249299
goat    x     A   -0.909280
              B    0.184450
        y     A   -0.755891
              B   -0.957222
dtype: float64
```

```
s = df.stack()
s['bird']['x']['A']
```

```
0.6997320218227948
```

Pivoting and Stacking

```
columns = pd.MultiIndex.from_tuples(
    [('A', 'cat', 'long'), ('B', 'cat', 'long'),
     ('A', 'dog', 'short'), ('B', 'dog', 'short')],
    names=['cond', 'animal', 'hair_length'])
df = pd.DataFrame(np.random.randn(4, 4), columns=columns)
df
```

cond	A	B	A	B
animal	cat	cat	dog	dog
hair_length	long	long	short	short
0	-0.424446	-0.204965	-2.494808	1.278635
1	-0.710625	-0.801063	0.947879	0.763564
2	0.016435	0.701775	-0.577844	-1.315433
3	0.451242	0.886683	-0.864094	0.529257

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.

cond	A	B	A	B
animal	cat	cat	dog	dog
hair_length	long	long	short	short
0	-0.424446	-0.204965	-2.494808	1.278635
1	-0.710625	-0.801063	0.947879	0.763564
2	0.016435	0.701775	-0.577844	-1.315433
3	0.451242	0.886683	-0.864094	0.529257

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

	cond	A		B	
	hair_length	long	short	long	short
	animal				
0	cat	-0.424446	NaN	-0.204965	NaN
	dog	NaN	-2.494808	NaN	1.278635
1	cat	-0.710625	NaN	-0.801063	NaN
	dog	NaN	0.947879	NaN	0.763564
2	cat	0.016435	NaN	0.701775	NaN
	dog	NaN	-0.577844	NaN	-1.315433
3	cat	0.451242	NaN	0.886683	NaN
	dog	NaN	-0.864094	NaN	0.529257

Pivoting and Stacking

cond	A	B	A	B
animal	cat	cat	dog	dog
hair_length	long	long	short	short
0	-0.424446	-0.204965	-2.494808	1.278635
1	-0.710625	-0.801063	0.947879	0.763564
2	0.016435	0.701775	-0.577844	-1.315433
3	0.451242	0.886683	-0.864094	0.529257

```
df.stack(level=[0,1,2])
```

```
   cond  animal  hair_length
0  A     cat     long           -0.424446
         dog     short          -2.494808
   B     cat     long           -0.204965
         dog     short           1.278635
1  A     cat     long           -0.710625
         dog     short           0.947879
   B     cat     long           -0.801063
         dog     short           0.763564
2  A     cat     long            0.016435
         dog     short          -0.577844
   B     cat     long            0.701775
         dog     short          -1.315433
3  A     cat     long            0.451242
         dog     short          -0.864094
   B     cat     long            0.886683
         dog     short           0.529257
dtype: float64
```

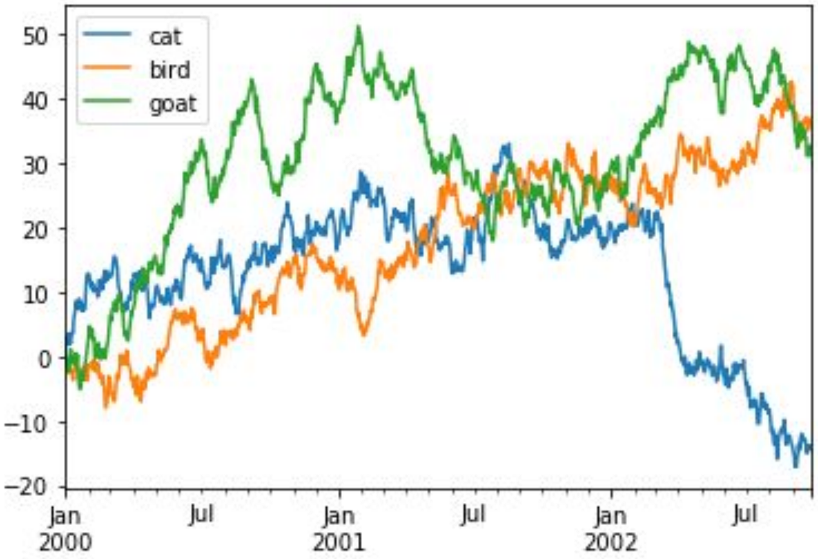
Stacking across all three levels yields a Series, since there is no longer any column structure. This is often called **flattening** a table.

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

```
df = pd.DataFrame(np.random.randn(1000, 3),
                  index=pd.date_range('1/1/2000', periods=1000),
                  columns=['cat','bird','goat'])
df = df.cumsum()
_ = df.plot()
```

`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.