STATS 507
Data Analysis in Python

Week10-1: Pandas and Seaborn

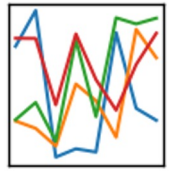
Dr. Xian Zhang

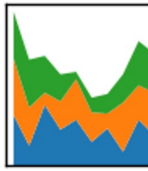
Adapted from slides by Professor Jeffrey Regier

Recap: Pandas

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$





Yet another **open-sourced**, **practical**, modern data science tool in Python…

- **Database-like structures**, largely similar to those available in R
- Well integrated with `numpy/scipy`
- Supports read/write for a wide range of different file format.
- Flexible and efficient

https://pandas.pydata.org/docs/user_guide/cookbook.html#cookbook

Recap Series Basics

`Pandas` series are Pandas data structure built on top of NumPy arrays.

- Series also contain a **custom index** and an **optional name**, in addition to the array of data.
- They can be created from other data types, but are usually imported form external sources (csv, Excel or SQL database).
- Two or more Series grouped together form a `Pandas` DataFrame.

Creation

```
1 import pandas as pd
2 import numpy as np
3 numbers = np.random.randn(5)
4 s = pd.Series(numbers)
5 s
```

```
0   -0.318743
1    0.807948
2   -0.216362
3   -0.356014
4    1.542122
dtype: float64
```

Indexing/Slicing

```
1 s = pd.Series([2,3,5,7,11])
2 s[0]
```

```
2
```

```
1 s[1:3]
```

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

```
1 s[-1]
```

Apply functions

```
1 s + 2*s
```

```
dog                  9.4245
cat                     126
bird                      0
goat                  4.854
cthulu    abcdeabcdeabcde
dtype: object
```

Universal functions and methods

Recap Dataframe Basics

Dataframes are fundamental unit of `pandas`

- 2-d structure (i.e., rows and columns).
- **Columns**, of potentially different datatypes (think, spreadsheet or database)
- Can be created from many different **objects** (supports read/write for a wide range of different file formats: .csv, SQL, SAS, HTML…) A major advantage!

Creation

```
df = pd.read_csv("forestfires.csv")
df
```

	X	Y	month	day	FFMC	DMC	DC	ISI	temp	RH	wind	rain	area
0	7	5	mar	fri	86.2	26.2	94.3	5.1	8.2	51	6.7	0.0	0.00
1	7	4	oct	tue	90.6	35.4	669.1	6.7	18.0	33	0.9	0.0	0.00
2	7	4	oct	sat	90.6	43.7	686.9	6.7	14.6	33	1.3	0.0	0.00
3	8	6	mar	fri	91.7	33.3	77.5	9.0	8.3	97	4.0	0.2	0.00
4	8	6	mar	sun	89.3	51.3	102.2	9.6	11.4	99	1.8	0.0	0.00
...	...	...	...	...	...	...	...	...	...	...	...	...	...
512	4	3	aug	sun	81.6	56.7	665.6	1.9	27.8	32	2.7	0.0	6.44
513	2	4	aug	sun	81.6	56.7	665.6	1.9	21.9	71	5.8	0.0	54.29
514	7	4	aug	sun	81.6	56.7	665.6	1.9	21.2	70	6.7	0.0	11.16
515	1	4	aug	sat	94.4	146.0	614.7	11.3	25.6	42	4.0	0.0	0.00
516	6	3	nov	tue	79.5	3.0	106.7	1.1	11.8	31	4.5	0.0	0.00

517 rows × 13 columns

Indexing/Slicing

```
1 presidents.loc['Obama']
```

```
1 presidents.iloc[1]
```

```
1 presidents[1:3]
```

```
1 presidents[presidents['Terms']<2]
```

Apply functions

```
1 df.apply(np.mean)
```

```
A   -0.293978
B    0.465070
C   -0.456450
D         NaN
dtype: float64
```

More operations to manipulate a `Pandas` Dataframe Object?

1. Reorganizing data: Group By

2. Statistical and other Computations

3. Stacking and pivoting

4. Visualize: Seaborn

Group By: reorganizing data

“Group By” operations are a concept from **databases**

1) Splitting data based on some criteria
2) Applying functions to different splits
3) 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.

What info should this object contain?

```
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

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

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

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

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

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

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

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.

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

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

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

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

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.

RESULT?

1) Aggregation allows you to perform calculations across rows to get summary statistics.
2) The aggregation is for each column.

https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.aggregate.html

Group By: hierarchically-indexed aggregation

```
1 arrs = [['math', 'math', 'econ', 'econ', 'stats', 'stats'],
2         ['left', 'right', 'left', 'right', 'left', 'right']]
3 index = pd.MultiIndex.from_arrays(arrs, names=['major', 'handedness'])
4 s = pd.Series(np.random.randn(6), index=index)
5 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.

https://pandas.pydata.org/docs/user_guide/advanced.html

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
```

```
1 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.

What should I group by?

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/pick up groups

```
1 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
```

```
1 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'`.

Aggregation for DataFrame

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

		A	B
major	handedness		
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

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

	A	B
handedness		
left	1.75	-0.546088
right	1.75	-0.258584

The `aggregation` method

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

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

handedness	A	B
left	1.75	-0.546088
right	1.75	-0.258584

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

handedness	A	B
left	7	-2.184352
right	7	-1.034337

Aggregating data

```
1 tsdf = pd.DataFrame(np.random.randn(10, 3),
2                     columns=['DOW', 'NASDAQ', 'S&P500'],
3                     index=pd.date_range('1/1/2000', periods=10))
4 tsdf.head()
```

	DOW	NASDAQ	S&P500
2000-01-01	1.118903	0.317094	-0.936392
2000-01-02	1.091083	0.828543	-1.961891
2000-01-03	-1.309894	-1.052207	0.256100
2000-01-04	0.654260	-0.527830	0.030650
2000-01-05	-1.041396	-0.559097	0.876613

Supplying a list of functions to `agg` will apply each function to each column of the DataFrame, with each function getting a row in the resulting DataFrame.

```
1 tsdf.agg([np.median, np.mean, np.std])
```

	DOW	NASDAQ	S&P500
median	0.534165	0.230327	-0.076018
mean	0.391512	0.159331	-0.239343
std	1.163320	0.907218	0.773417

`agg` is an alias for the method `aggregate`. Both work exactly the same.

Aggregating data

	DOW	NASDAQ	S&P500
2000-01-01	1.118903	0.317094	-0.936392
2000-01-02	1.091083	0.828543	-1.961891
2000-01-03	-1.309894	-1.052207	0.256100
2000-01-04	0.654260	-0.527830	0.030650
2000-01-05	-1.041396	-0.559097	0.876613

`agg` can, alternatively, take a dictionary whose keys are column names, and values are functions.

```
1 tsdf.agg({'DOW':'mean',
2           'NASDAQ':'median',
3           'S&P500':'max'})
```

Note that the values here are strings, not functions! `pandas` supports dispatch on strings. It recognizes certain strings as referring to functions. `apply` supports similar behavior.

```
NASDAQ     0.230327
S&P500     0.876613
DOW        0.391512
dtype: float64
```

Aggregating data for mixed data types

```
1 df = pd.DataFrame({'A': [1, 2, 3],
2                    'B': [1., 2., 3.],
3                    'C': ['foo', 'bar', 'baz']})
4 df
```

	A	B	C
0	1	1.0	foo
1	2	2.0	bar
2	3	3.0	baz

df now contained mixed data types…

`agg` (and similarly apply) will only try to apply these functions on the columns of types supported by those functions.

```
1 df.agg(['mean', 'max'])
```

	A	B	C
max	3.0	3.0	foo
mean	2.0	2.0	NaN

`pandas` doesn't know how to compute a mean string, so it doesn't try.

In class practice

1. Reorganizing data: Group By

2. Statistical and other Computations

3. Stacking and pivoting

4. Visualize: Seaborn

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.

Note: `pandas` has extensive support for **time series data**, which we mostly won't talk about in this course. Refer to the documentation/cookbook for more.

```
1 s = pd.Series(np.random.randn(8))
2 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
```

```
1 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 rows 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.

	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

`pct_change` includes control over how missing data is imputed, etc. See documentation for more detail: 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.

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

```
0.1522727637202401
```

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

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

	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

	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

Ranking method

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

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

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

```
1 s.rank()
```

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

Ties are broken (0.5) by assigning the mean rank to both values.

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

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

Ranking data

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

	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

```
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.

```
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**.

Iterating over Series and DataFrames

```
1 s
```

```
apple       fruit
cat        animal
goat       animal
banana      fruit
kiwi        fruit
dtype: object
```

```
1 for x in s:
2     print x
```

```
fruit
animal
animal
fruit
fruit
```

Iterating over a Series gets an iterator over the values of the Series.

Iterating over a DataFrame gets an iterator over the column names.

```
1 df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

```
1 for x in df:
2     print x
```

```
A
B
C
```

Iterating over Series and DataFrames

```
1 for x in df.iteritems():
2     print(x)
```

```
('A', 0   -2.072339
1   -0.587874
2   -0.164436
3   -1.215576
4   -0.350299
Name: A, dtype: float64)
('B', 0   -1.282539
1    0.517591
2    1.450398
3   -0.671235
4    1.958805
Name: B, dtype: float64)
('C', 0   -1.241128
1   -0.394561
2   -0.975424
3    0.394053
4    0.467778
Name: C, dtype: float64)
```

`iteritem()` method is supported by both Series and DataFrames. Returns an iterator over the **key-value** pairs. In the case of Series, these are (index,value) pairs. In the case of DataFrames, these are (colname, Series) pairs.

```
1 df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

Iterating over Series and DataFrames

```
1 for x in df.iterrows():
2     print(x)
```

```
(0, A   -2.072339
B   -1.282539
C   -1.241128
Name: 0, dtype: float64)
(1, A   -0.587874
B    0.517591
C   -0.394561
Name: 1, dtype: float64)
(2, A   -0.164436
B    1.450398
C   -0.975424
Name: 2, dtype: float64)
(3, A   -1.215576
B   -0.671235
C    0.394053
Name: 3, dtype: float64)
(4, A   -0.350299
B    1.958805
C    0.467778
Name: 4, dtype: float64)
```

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.

```
1 df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

Iterating over Series and DataFrames

```
1 for x in df.iterrows():
2     print(x)
```

```
(0, A   -2.072339
B   -1.282539
C   -1.241128
Name: 0, dtype: float64)
(1, A   -0.587874
B    0.517591
C   -0.394561
Name: 1, dtype: float64)
(2, A   -0.164436
B    1.450398
C   -0.975424
Name: 2, dtype: float64)
(3, A   -1.215576
B   -0.671235
C    0.394053
Name: 3, dtype: float64)
(4, A   -0.350299
B    1.958805
C    0.467778
Name: 4, dtype: float64)
```

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.

```
1 df
```

4	-0.350299	1.958805	0.467778

Note: DataFrames are designed to make certain operations (mainly **vectorized operations**) fast. This implementation has the disadvantage that iteration over a DataFrames is slow. It is usually best to avoid iterating over the elements of a DataFrame or Series, and instead find a way to compute your quantity of interest using a vectorized operation or a map/reduce operation.

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."

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

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

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

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

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

Like `transform`, the result is ungrouped.

Combining DataFrames

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

`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.

	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

Missing values get NaN.

1. Reorganizing data: Group By

2. Statistical Computation

3. Stacking and pivoting

4. Visualize: Seaborn

Motivation for unstack

	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.

What DataFrame we might want?

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

Use pivot to unstack

	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.

```
1 df.pivot(index='date',
2          columns='variable',
3          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

The reverse process

```
1 tuples = list(zip(*[['bird','bird','goat','goat'],
2                     ['x', 'y', 'x', 'y']]))
3 index = pd.MultiIndex.from_tuples(tuples,names=['animal','cond'])
4 df = pd.DataFrame(np.random.randn(4, 2),
5                   index=index, columns=['A', 'B'])
6 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.

Stacking

		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

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.

```
1 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
```

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

```
0.6997320221822794
```

Pivoting and Stacking

```
1 columns = pd.MultiIndex.from_tuples(
2     [('A', 'cat', 'long'), ('B', 'cat', 'long'),
3      ('A', 'dog', 'short'), ('B', 'dog', 'short')],
4     names=['cond', 'animal', 'hair_length'])
5 df = pd.DataFrame(np.random.randn(4, 4), columns=columns)
6 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.

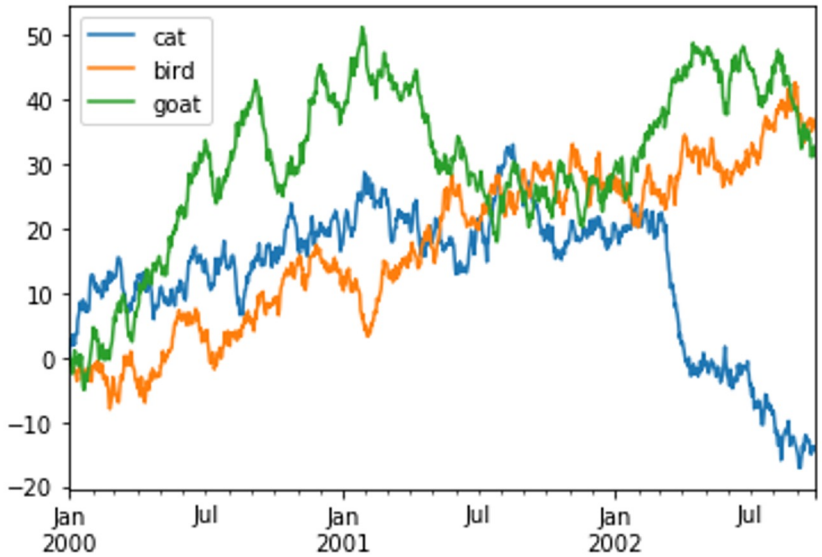
cond	A	B	A	B
animal	cat	cat	dog	dog
hair_length	long	long	short	short
0	-0.424446	-0.204965	-2.494808	1.278635
1	-0.710625	-0.801063	0.947879	0.763564
2	0.016435	0.701775	-0.577844	-1.315433
3	0.451242	0.886683	-0.864094	0.529257

```
1 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

Can Plotting DataFrames Directly

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

1. Reorganizing data: Group By

2. Statistical Computation

3. Stacking and pivoting

4. Visualize: One Seaborn examples

Exploratory Data Analysis (EDA)

Typical steps that a Data Analyst might take:

- Performs initial analysis to assess the quality of the data.
- Removes dirty data as and when discovered.
- Performs Exploratory Data Analysis (EDA) to determine the meaning of the data.
- Presents the findings with Explanatory Diagrams, Summaries and Narratives to the management.

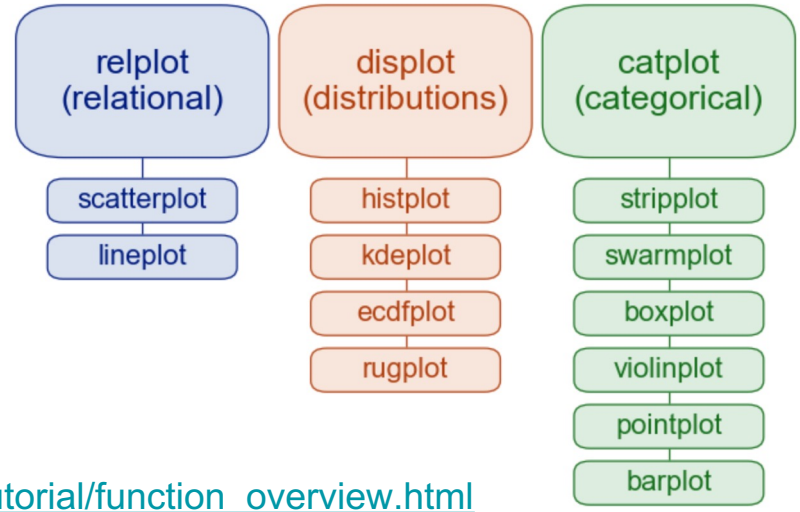
Typical initial questions to start with:

- How data is distributed?
- How variables are correlated?

Seaborn

Seaborn is a library for making **statistical graphics** in Python. It builds on top of matplotlib and integrates closely with pandas data structures.

- A high-level database oriented API
- Most of your interactions with seaborn will happen through a set of **plotting** functions



https://seaborn.pydata.org/tutorial/function_overview.html

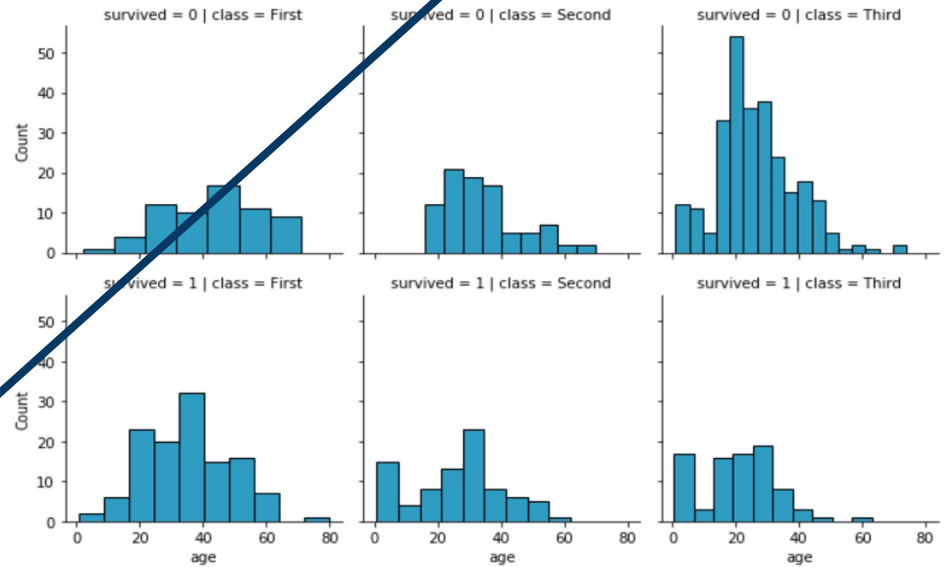
Seaborn example1: facetgrid

Multi-plot grid for plotting conditional relationships.

Seaborn library comes with many datasets commonly used in data science teaching and you can simply use them by using `load_dataset`

‘row’ and ‘col’ keyword arguments expect the column names to use from the given dataframe

```
import seaborn as sns
titanic = sns.load_dataset('titanic')
grid = sns.FacetGrid(titanic, row='survived', col='class')
_ = grid.map(sns.histplot, "age")
```



Other things

HW7 is out

We will expect you learn LineCharts, BarCharts and CatPlot on your own for HW7. Read the tutorials!

Coming next:

- Regular expressions in Python and SQL
- Final project guideline (next Monday)