Introduction to Data Structures

Pandas
in Python

Begin from the Beginning

- Pandas is used typically along with numpy

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

- Two important data structures

  `pd.Series, pd.DataFrame`

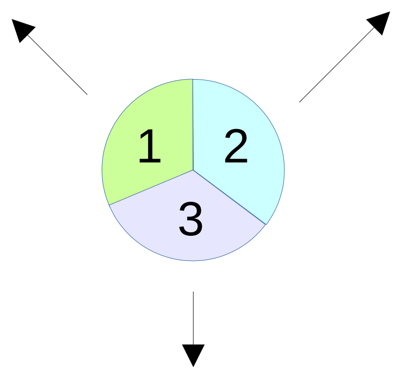
- The data are labeled, and the link between will not be broken unless explicitly done so. That is the *data alignment is intrinsic*.

DataFrame from dict of Series

```
d = {'one': pd.Series([1, 2, 3],    index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 
'd'])}
```

```
df = pd.DataFrame(d)    pd.DataFrame(d, index=['d', 'b', 'a'])
```

```
In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```




```
Out[40]:
one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
```

```
pd.DataFrame(d, index=['d', 'b', 'a'],
                columns=['two', 'three'])

Out[41]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

DataFrame from dict of ndarrays

- The ndarrays must all be the same length.
- If an index is passed, it must clearly also be the same length as the arrays.
- If no index is passed, the result will be range(n), where n is the array length.

```
d = {'one': [1., 2., 3., 4.],
     'two': [4., 3., 2., 1.]}
```

```
pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0
```

```
pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

DataFrame from record array

```
data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])
data[:] = [(1, 2., 'Hello'), (2, 3., "World")]
```

```
pd.DataFrame(data)
Out[49]:
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'
```

```
pd.DataFrame(data, index=['first', 'second'])
Out[50]:
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'
```

```
pd.DataFrame(data, columns=['C', 'A', 'B'])
Out[51]:
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

DataFrame from list of dicts

```
In [52]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

pd.DataFrame(data2)
Out[53]:
   a   b     c
0  1   2   NaN
1  5  10  20.0

pd.DataFrame(data2, index=['first', 'second'])
Out[54]:
        a   b     c
first   1   2   NaN
second  5  10  20.0

pd.DataFrame(data2, columns=['a', 'b'])
Out[55]:
   a   b
0  1   2
1  5  10
```

DataFrame from dict of objects

```
In [9]: df2 = pd.DataFrame({'A': 1.,
   ...:                     'B': pd.Timestamp('20130102'),
   ...:                     'C': pd.Series(1, index=list(range(4)), dtype='float32'),
   ...:                     'D': np.array([3] * 4, dtype='int32'),
   ...:                     'E': pd.Categorical(["test", "train", "test", "train"]),
   ...:                     'F': 'foo'})
   ...:

In [10]: df2
Out[10]:
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

```
In [11]: df2.dtypes
Out[11]:
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

DataFrame from Series

- The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

Alternate constructors

```
pd.DataFrame.from_dict(dict([('A', [1, 2, 3]),
                             ('B', [4, 5, 6])]))

Out[57]:

   A  B

0  1  4

1  2  5

2  3  6

pd.DataFrame.from_dict(dict([('A', [1, 2, 3]),
                             ('B', [4, 5, 6])]),orient='index',
                             columns=['one', 'two', 'three'])

Out[58]:

   one  two  three

A    1    2      3

B    4    5      6
```

Alternate constructors

```
data = array([(1, 2., b'Hello'),
              (2, 3., b'World')],
   dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

pd.DataFrame.from_records(data, index='C')
Out[60]:
          A    B
C
b'Hello'  1  2.0
b'World'  2  3.0
```

Column selection, addition, deletion

```
In [62]: df['three'] = df['one'] * df['two']

In [63]: df['flag'] = df['one'] > 2

In [64]: df

Out[64]:

   one  two  three   flag

a  1.0  1.0    1.0  False

b  2.0  2.0    4.0  False

c  3.0  3.0    9.0   True

d  NaN  4.0    NaN  False
```

```
In [61]: df['one']

Out[61]:

a    1.0

b    2.0

c    3.0

d    NaN

Name: one, dtype:
float64
```

More del/ins operations with df

Columns can be deleted or popped like with a dict:

```
del df['two']

three = df.pop('three')

In [67]: df
Out[67]:
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame’s index:

```
df['one_trunc'] = df['one'][:2]

In [71]: df
Out[71]:
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

Insert method of DataFrame

- You can insert raw ndarrays but their length must match the length of the DataFrame’s index.
- By default, columns get inserted at the end.
- The insert function is available to insert at a particular location in the columns:

```
In [72]: df.insert(1, 'bar', df['one'])

In [73]: df
Out[73]:
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Assign new cols in method chains

```
In [74]: iris = pd.read_csv('data/iris.data')
In [75]: iris.head()
Out[75]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

In [76]: (iris.assign(sepal_ratio=iris['SepalWidth'] / iris['SepalLength']).head())
Out[76]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

- DataFrame's `assign()` method is inspired by dplyr's mutate verb, that allows you to easily create new columns from existing columns; leaves orig dataframe unmodified.

Assign using lambda

Pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```
In [77]: iris.assign(sepal_ratio=lambda x: (x['SepalWidth'] /
x['SepalLength'])).head() # iris is renamed as x in the lambda, redundant??

Out[77]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

Chaining methods

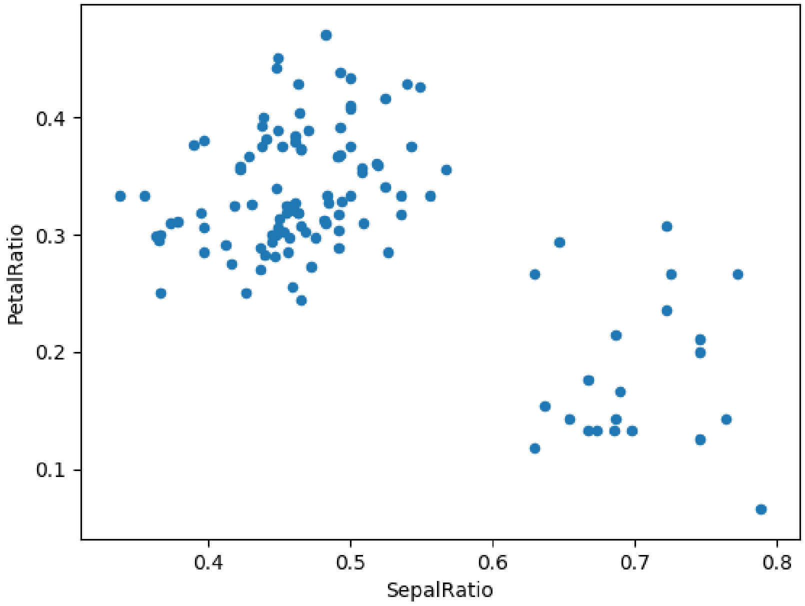
Passing a callable, as opposed to an actual value to be inserted, is useful when you don't have a reference to the DataFrame at hand.

This is common when using assign in a chain of operations.

For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```
In [78]: (iris.query('SepalLength > 5')
   ....:       .assign(SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ....:               PetalRatio=lambda x: x.PetalWidth / x.PetalLength)
   ....:       .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
```

.



Indexing / selection

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Select rows by boolean vector	`df[bool_vec]`	DataFrame

Single row selection

- Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [83]: df.loc['b']
Out[83]:
one                  2
bar                  2
flag             False
foo                bar
one_trunc            2
Name: b, dtype: object
```

```
In [84]: df.iloc[2]
Out[84]:
one                  3
bar                  3
flag              True
foo                bar
one_trunc          NaN
Name: c, dtype: object
```

Data Alignment

```
df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
In [87]: df + df2
Out[87]:
          A         B         C    D
0  0.045691 -0.014138  1.380871  NaN
1 -0.955398 -1.501007  0.037181  NaN
2 -0.662690  1.534833 -0.859691  NaN
3 -2.452949  1.237274 -0.133712  NaN
4  1.414490  1.951676 -2.320422  NaN
5 -0.494922 -1.649727 -1.084601  NaN
6 -1.047551 -0.748572 -0.805479  NaN
7       NaN       NaN       NaN  NaN
8       NaN       NaN       NaN  NaN
9       NaN       NaN       NaN  NaN
```

```
In [88]: df – df.iloc[0] # Row broadcast
Out[88]:
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1 -1.359261 -0.248717 -0.453372 -1.754659
2  0.253128  0.829678  0.010026 -1.991234
3 -1.311128  0.054325 -1.724913 -1.620544
4  0.573025  1.500742 -0.676070  1.367331
5 -1.741248  0.781993 -1.241620 -2.053136
6 -1.240774 -0.869551 -0.153282  0.000430
7 -0.743894  0.411013 -0.929563 -0.282386
8 -1.194921  1.320690  0.238224 -1.482644
9  2.293786  1.856228  0.773289 -1.446531
```

```
In [20]: row = df.iloc[1], In [21]: column = df['B']
df.sub(row, axis='columns') == df.sub(row, axis=1)  # Row Broadcast
df.sub(column, axis='index') == df.sub(column, axis=0)  # Column Broadcast
```

DataFrame from a NumPy array, datetime index, & labeled columns

```
dates = pd.date_range('20130101', periods=6) # YYYY MM DD

In [6]: dates

Out[6]:

DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03',
'2013-01-04','2013-01-05', '2013-01-06'],
dtype='datetime64[ns]', freq='D')

df = pd.DataFrame(np.random.randn(6, 4), index=dates,
columns=list('ABCD'))

In [8]: df

Out[8]:                    A         B         C         D
        2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
        2013-01-02  1.212112 -0.173215  0.119209 -1.044236
        2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
        2013-01-04  0.721555 -0.706771 -1.039575  0.271860
        2013-01-05 -0.424972  0.567020  0.276232 -1.087401
        2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Functions, Operators, Reductions

```
df1 = pd.DataFrame({'a': [1, 0, 1], 'b': [0, 1, 1]}, dtype=bool)
df2 = pd.DataFrame({'a': [0, 1, 1], 'b': [1, 1, 0]}, dtype=bool)
dfs = pd.Series(np.random.randn(1000))
df1 & df2, df1 | df2, df1 ^ df2, - df1,
df1.gt/lt/ge/le/ne/eq(df2) # elementwise
(df1 > 0).all/any() # columnwise reductions
df1.equals(df2) # True / False, treats nan=nan as True, unlike df1==df2
df1.combibe_first(df2) # substitution of Nan in df1 from df2
df1.mean(0) is colmeans, df1.mean(1) is row means.
df.sum(axis=1, skipna=True)
(df - df.mean()) / df.std(), df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)
dfs.describe(percentiles=[.05, .25, .75, .95]) # try with and w/o percentiles
df1.idxmin/idxmax(axis=0/1) # index of min and max
df1[:5].T # Transpose
df.sort_values(by=column_label)
df.loc[start_row:end_row, ['A', 'B']] # A, B are sample column list
df.iloc[[1, 2, 4], [0, 2]] #row list, followed by col list
```

DataFrame interoperability with NumPy functions

```
np.exp(df) #all ufuncs applicable, log, sin, sqrt
```

isin method of DataFrame

```
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
In [41]: df2 = df.copy()

In [42]: df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']

In [43]: df2
Out[43]:
                   A         B         C         D      E
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632    one
2013-01-02  1.212112 -0.173215  0.119209 -1.044236    one
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804    two
2013-01-04  0.721555 -0.706771 -1.039575  0.271860  three
2013-01-05 -0.424972  0.567020  0.276232 -1.087401   four
2013-01-06 -0.673690  0.113648 -1.478427  0.524988  three

In [44]: df2[df2['E'].isin(['two', 'four'])]
Out[44]:
                   A         B         C         D     E
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804   two
2013-01-05 -0.424972  0.567020  0.276232 -1.087401  four
```

More methods

- `df1.dropna(how='any') #drop rows having nan`
- `df1.fillna(value=5) #presets for nan`
- `pd.isna(df1) #bool matrix`
- `df.apply(np.cumsum)`
- `df.apply(lambda x: x.max() - x.min())`
- `pieces = [df[:3], df[3:7], df[7:]]`
- `pd.concat(pieces) #get back df`

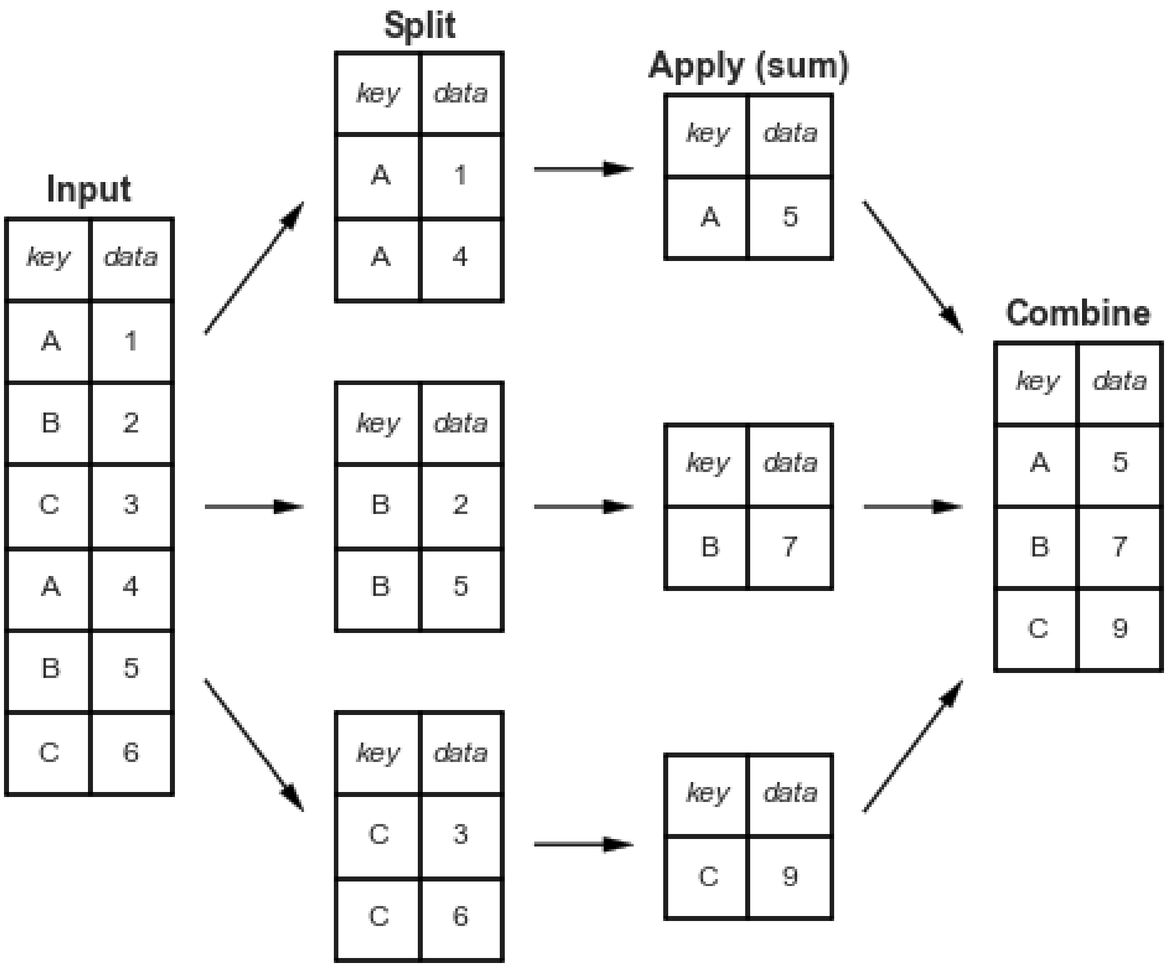
Join as in SQL

```
In [82]: left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
In [83]: right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})

In [84]: left           In [85]: right
Out[84]:                Out[85]:
   key  lval              key  rval
0  foo     1            0  foo     4
1  bar     2            1  bar     5

In [86]: pd.merge(left, right, on='key')
Out[86]:
   key  lval  rval
0  foo     1     4
1  bar     2     5
```

Split, Apply, Combine ~ Groupby + Aggregate



Groupby as in SQL

```
df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'foo'],
   ....:            'B': ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three'],
   ....:            ‘C': np.random.randn(8),'D': np.random.randn(8)})
   ....:
```

```
In [92]: df
Out[92]:
     A      B         C         D
0  foo    one -1.202872 -0.055224
1  bar    one -1.814470  2.395985
2  foo    two  1.018601  1.552825
3  bar  three -0.595447  0.166599
4  foo    two  1.395433  0.047609
5  bar    two -0.392670 -0.136473
6  foo    one  0.007207 -0.561757
7  foo  three  1.928123 -1.623033
```

```
df.groupby(['A', 'B']).sum()
Out[94]:
                  C         D
A   B
bar one   -1.814470  2.395985
    three -0.595447  0.166599
    two   -0.392670 -0.136473
foo one   -1.195665 -0.616981
    three  1.928123 -1.623033
    two    2.414034  1.600434
```

```
df.groupby('A').sum()
Out[93]:
            C        D
A
bar -2.802588  2.42611
foo  3.146492 -0.63958
```

Pivot Tables

When users create a pivot table, there are four main components:

- Columns- When a field is chosen for the column area, only the unique values of the field are listed across the top.
- Rows- When a field is chosen for the row area, it populates as the first column. Similar to the columns, all row labels are the unique values and duplicates are removed.
- Values- Each value is kept in a pivot table cell and display the summarized information. The most common values are sum, average, minimum and maximum.
- Filters- Filters apply a calculation or restriction to the entire table.

Pivot Tables

```
 df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo", "bar", "bar", "bar", "bar"],
                    "B": ["one", "one", "one", "two", "two", "one", "one", "two", "two"],
                    "C": ["small", "large", "large", "small", small", "large", "small",
"small", "large"],
                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
>>> df
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

```
>>> table
C           large  small
A   B
bar one       4.0    5.0
    two       7.0    6.0
foo one       4.0    1.0
    two       NaN    6.0
```

```
table = pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'], aggfunc=np.sum)
```

Pivot tables

```
table1 = pd.pivot_table(df,
values='D', index=['A', 'B'], columns=['C'],
aggfunc=np.sum, fill_value=0)
```

```
table2 = pd.pivot_table(df,
values=['D', 'E'], index=['A', 'C'],
aggfunc={'D': np.mean,'E': np.mean})
```

```
table3 = pd.pivot_table(df,
values=['D', 'E'], index=['A', 'C'],
aggfunc={'D': np.mean, 'E': [min, max, np.mean]})
```

```
>>> table3
              D        E
              mean     max    mean      min
A   C
bar large  5.500000  9.0  7.500000  6.0
    small  5.500000  9.0  8.500000  8.0
foo large  2.000000  5.0  4.500000  4.0
    small  2.333333  6.0  4.333333  2.0
```

Multiple aggregates for a value column

```
>>> table1
C          large  small
A   B
bar one        4      5
    two        7      6
foo one        4      1
    Two        0      6
```

Sum

```
>>> table2
                D          E
A   C
bar large  5.500000  7.500000
    small  5.500000  8.500000
foo large  2.000000  4.500000
    Small  2.333333  4.333333
```

Mean across multiple columns

Multi-level index pivot table

Earlier only one feature was used in the index, i.e., a single level index.

We can, however, create pivot tables using multiple indices.

Whenever data is organized hierarchically, a pivot table with multi-level indexes can provide very useful and detailed summary information.

```
table3 = pd.pivot_table(df,
   values=['D', 'E'],
   index=['A', 'C'],
   aggfunc={'D': np.mean, 'E':
[min, max, np.mean]})
```

```
>>> table3
```

		D	E		
		mean	max	mean	min
A	C				
bar	large	5.500000	9.0	7.500000	6.0
	small	5.500000	9.0	8.500000	8.0
foo	large	2.000000	5.0	4.500000	4.0
	small	2.333333	6.0	4.333333	2.0

Multiple aggregates for a value column

Groupby vs pivot_table

Both pivot_table and groupby are used to aggregate your dataframe. The difference is only with regard to the shape of the result.

Using groupby, the dimensions given are placed into columns, and rows are created for each combination of those dimensions.

pivot_table = groupby + unstack

groupby = pivot_table + stack

In particular, if columns parameter of pivot_table() is not used, then groupby() and pivot_table() both produce the same result (if the same aggregator function is used).

Read from stackoverflow

Method Chaining

- A pointed example for method chaining can be seen here. A must read one.
- http://tomaugspurger.github.io/method-chaining.html

I/o in pandas

- See https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-excel-reader

Works

- Get airport data from http://ourairports.com/data/

```
import pandas as pd
airports = pd.read_csv('data/airports.csv')
airport_freq = pd.read_csv('data/airport-frequencies.csv')
runways = pd.read_csv('data/runways.csv')
```

SQL	Pandas
select * from airports	airports
select * from airports limit 3	airports.head(3)
select distinct type from airport	airports.type.unique()
select id from airports where ident = 'KLAX'	airports[airports.ident == 'KLAX'].id

Works – Where, Select, Orderby

Filters-

```
select * from airports where iso_region = 'US-CA' and type =
'seaplane_base'
```

```
airports[(airports.iso_region == 'US-CA') & (airports.type ==
'seaplane_base')]
```

Filter and Choose columns -

```
select ident, name, municipality from airports where iso_region = 'US-CA'
and type = 'large_airport'
```

```
airports[(airports.iso_region == 'US-CA') & (airports.type ==
'large_airport')][['ident', 'name', 'municipality']]
```

Ordering -

```
select * from airport_freq where airport_ident = 'KLAX' order by type
```

```
airport_freq[airport_freq.airport_ident == 'KLAX'].sort_values('type')
```

```
select * from airport_freq where airport_ident = 'KLAX' order by type desc
```

```
airport_freq[airport_freq.airport_ident == 'KLAX'].sort_values('type',
ascending=False) |
```

Having

- ```
  select type, count() from airports
  where iso_country = 'US' group by type
  having count() > 1000 order by count()
  desc
  ```
- ```
  airports[airports.iso_country == 'US']
      .groupby('type')
      .filter(lambda g: len(g) > 1000)
      .groupby('type')
      .size()
      .sort_values(ascending=False)
  ```

Grouby, Count, Orderby

- `select iso_country, type, count() from airports group by iso_country, type order by iso_country, type`
- `airports.groupby(['iso_country', 'type']).size()`
- `select iso_country, type, count() from airports group by iso_country, type order by iso_country, count() desc`
- ```
  airports.groupby(['iso_country', 'type'])
          .size()
          .to_frame('size')
          .reset_index()
          .sort_values(['iso_country', 'size'], ascending=[True, False])
  ```
  ```

JOIN / merge revisited

- Need to provide which columns to join on (left_on and right_on), and join type: inner (default), left (corresponds to LEFT OUTER in SQL), right (RIGHT OUTER), or outer (FULL OUTER).
- `select airport_ident, type, description, frequency_mhz from airport_freq join airports on airport_freq.airport_ref = airports.id where airports.ident = 'KLAX'`
- `airport_freq.merge(airports[airports.ident == 'KLAX'][['id']], left_on='airport_ref', right_on='id', how='inner')[['airport_ident', 'type', 'description', 'frequency_mhz']]`

Insert / concat

- There’s no such thing as an INSERT in Pandas. Instead, you would create a new dataframe containing new records, and then concat the two
- `create table heroes (id integer, name text)`
- `insert into heroes values (1, 'Harry Potter')`
- `insert into heroes values (2, 'Ron Weasley');`
- `df1 = pd.DataFrame({'id': [1, 2], 'name': ['Harry Potter', 'Ron Weasley']})`
- `insert into heroes values (3, 'Hermione Granger')`
- `df2 = pd.DataFrame({'id': [3], 'name': ['Hermione Granger']})`
- `pd.concat([df1, df2]).reset_index(drop=True)`

Union / Concat

- Use pd.concat() to UNION ALL two dataframes:
- `select name, municipality from airports where ident = 'KLAX'`

  ```
  union all
  ```

  `select name, municipality from airports where ident = 'KLGB'`
- `pd.concat([airports[airports.ident == 'KLAX'][['name', 'municipality']], airports[airports.ident == 'KLGB'][['name', 'municipality']]])`

UPDATE

- `update airports set home_link = 'http://www.lawa.org/welcomelax.aspx' where ident == 'KLAX'`
- `airports.loc[airports['ident'] == 'KLAX', 'home_link'] = 'http://www.lawa.org/welcomelax.aspx'`

DELETE / drop

- `delete from lax_freq where type = 'MISC'`
- The easiest (and the most readable) way to “delete” things from a Pandas dataframe is to subset the dataframe to rows you want to keep.
- `lax_freq = lax_freq[lax_freq.type != 'MISC']`
- Alternatively, you can get the indices of rows to delete, and .drop() rows using those indices:
- `lax_freq.drop(lax_freq[lax_freq.type == 'MISC'].index)`

Aggregate functions

- `select max(length_ft), min(length_ft), mean(length_ft), median(length_ft) from runways`
- `runways.agg({'length_ft': ['min', 'max', 'mean', 'median']})`

DataFrame

- DataFrame accepts different kinds of input:
  - Dict of 1D ndarrays, lists, dicts, or Series
  - 2-D numpy.ndarray
  - Structured or record ndarray
  - A Series
  - Another DataFrame