Python for Data Analytics

Pandas I

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 6 – 17, 2020

Outline

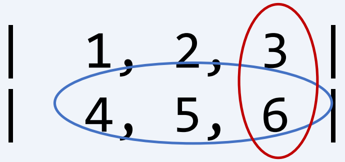
- Why Pandas?
- Pandas Series
- Pandas DataFrame
- I/O in Pandas
- Time Series Data in Pandas

Why Pandas?

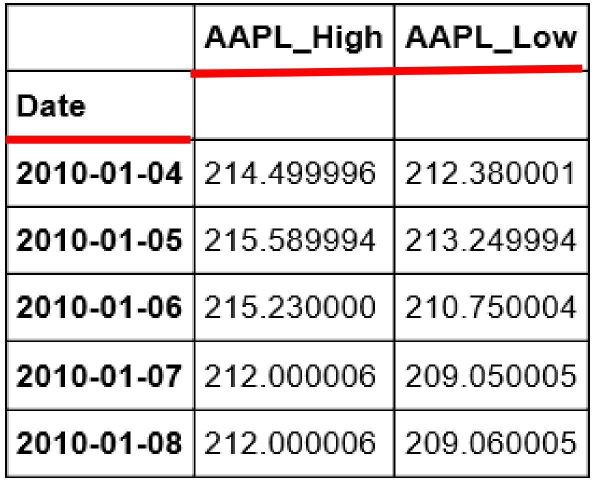
Limitations in NumPy

- Remember? Array slicing in NumPy

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a[1,:]
array([[4, 5, 6]])
>>> a[:,2]
array([3, 6])
>>> a[-1:,-2:]
array([[5, 6]])
```



How about?



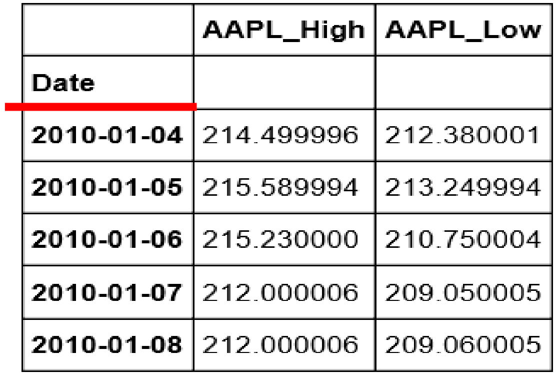
	AAPL_High	AAPL_Low
Date		
2010-01-04	214.499996	212.380001
2010-01-05	215.589994	213.249994
2010-01-06	215.230000	210.750004
2010-01-07	212.000006	209.050005
2010-01-08	212.000006	209.060005

2010-01-06 ~ 2010-01-07 사이에 발생한 data 추출?

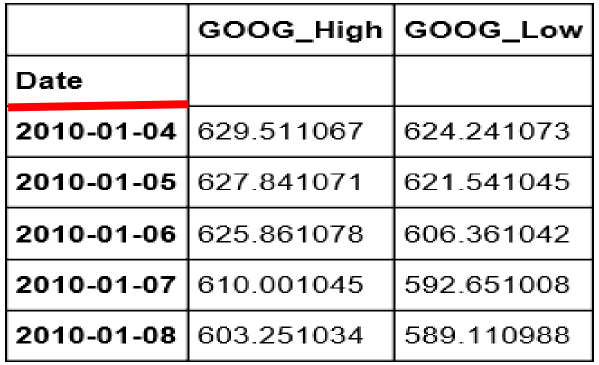
2010년에 월별로 발생한 data를 grouping?

Limitations in NumPy (cont'd)

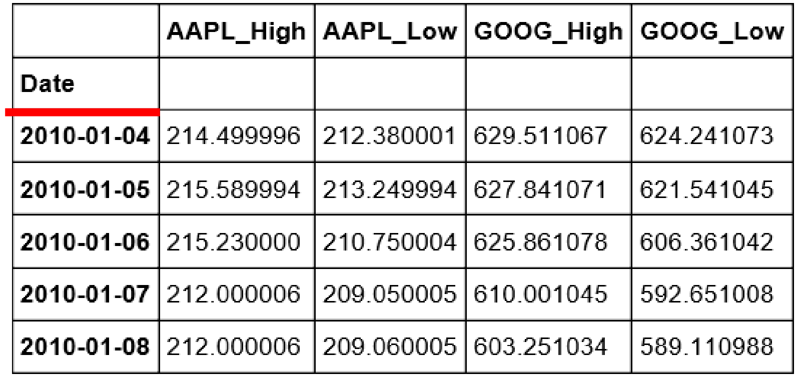
How about?



	AAPL_High	AAPL_Low
Date		
2010-01-04	214.499996	212.380001
2010-01-05	215.589994	213.249994
2010-01-06	215.230000	210.750004
2010-01-07	212.000006	209.050005
2010-01-08	212.000006	209.060005



	GOOG_High	GOOG_Low
Date		
2010-01-04	629.511067	624.241073
2010-01-05	627.841071	621.541045
2010-01-06	625.861078	606.361042
2010-01-07	610.001045	592.651008
2010-01-08	603.251034	589.110988



	AAPL_High	AAPL_Low	GOOG_High	GOOG_Low
Date				
2010-01-04	214.499996	212.380001	629.511067	624.241073
2010-01-05	215.589994	213.249994	627.841071	621.541045
2010-01-06	215.230000	210.750004	625.861078	606.361042
2010-01-07	212.000006	209.050005	610.001045	592.651008
2010-01-08	212.000006	209.060005	603.251034	589.110988

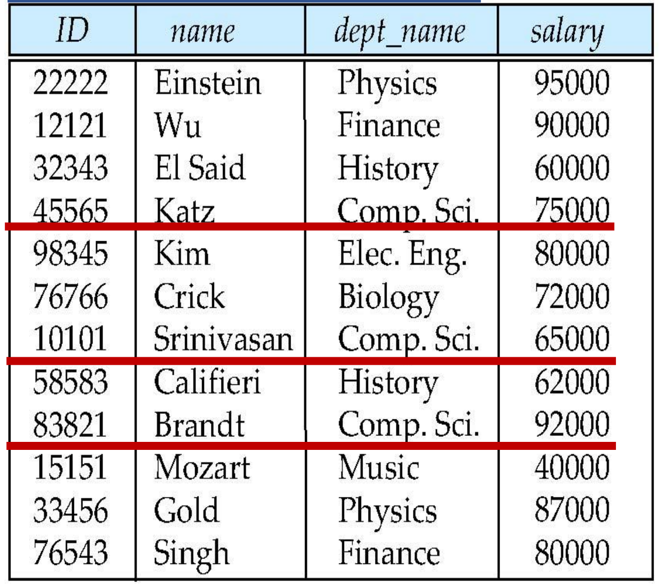
두 테이블의 join?

SQL and Tables (1)

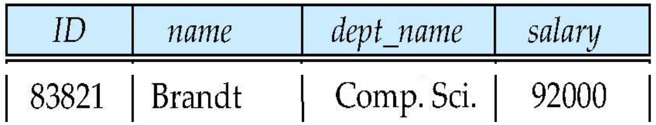
- Find all instructors in Comp. Sci. dept. with salary > 80000

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 80000;
```

Instructor relation



ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



ID	*name*	*dept_name*	*salary*
83821	Brandt	Comp. Sci.	92000

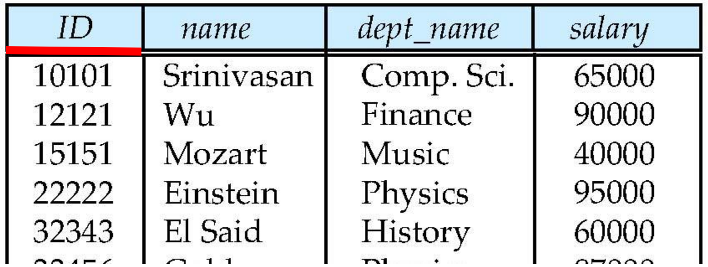
SQL and Tables (2)

- For all instructors who have taught courses, find their names and the course ID of the courses they taught

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

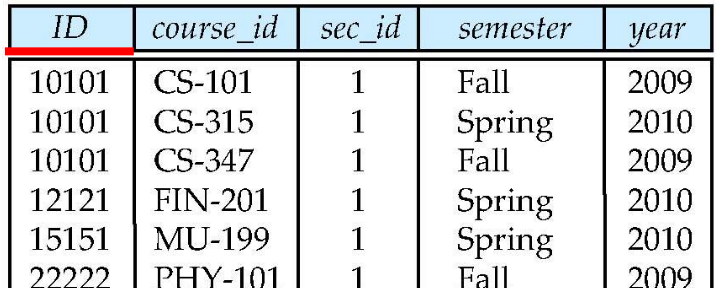
```
select *
from instructor natural join teaches;
```

instructor

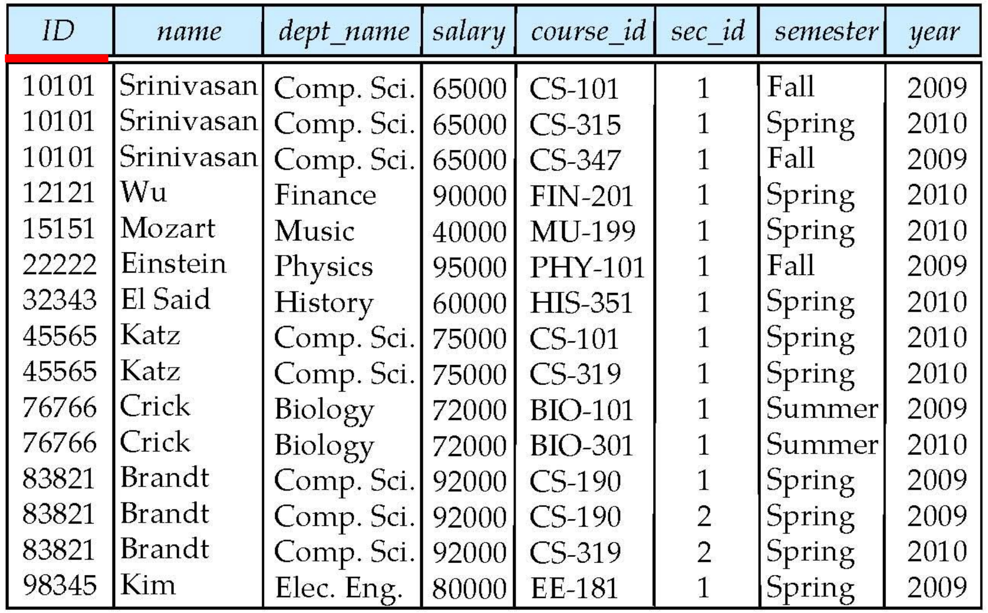


ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches



ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009



ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

instructor ⋈ *teaches*

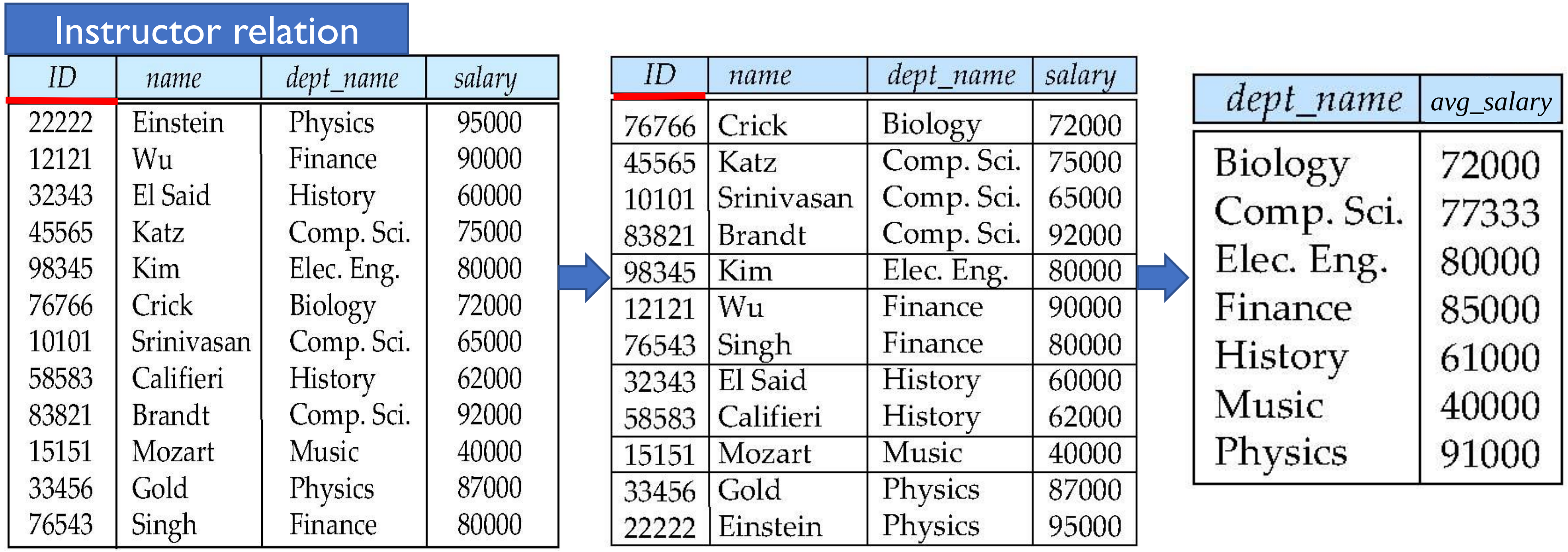
SQL and Tables (3)

- Group instructors in each department

```
select *
from instructor
group by dept_name;
```

- Find the average salary of instructors in each department

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name;
```



Instructor relation

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

ID	name	dept_name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Data Collection for Data Analytics

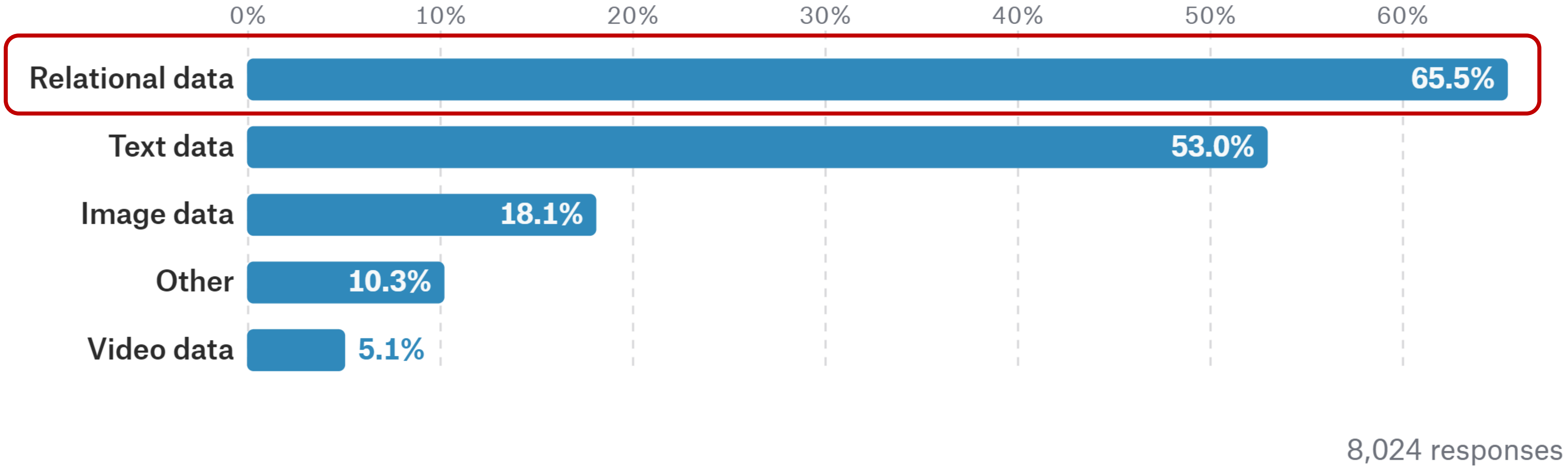
- You will typically get data in one of four ways:

1. Directly download a data file (or files) manually
2. Query data from a database
3. Query an API (usually web-based, these days)
4. Scrap data from a webpage

How to perform data preprocessing in Python?

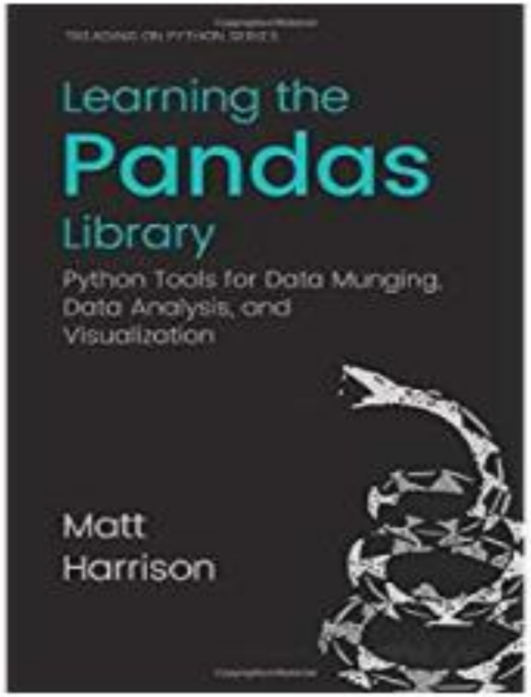
Data Format in Data Analytics

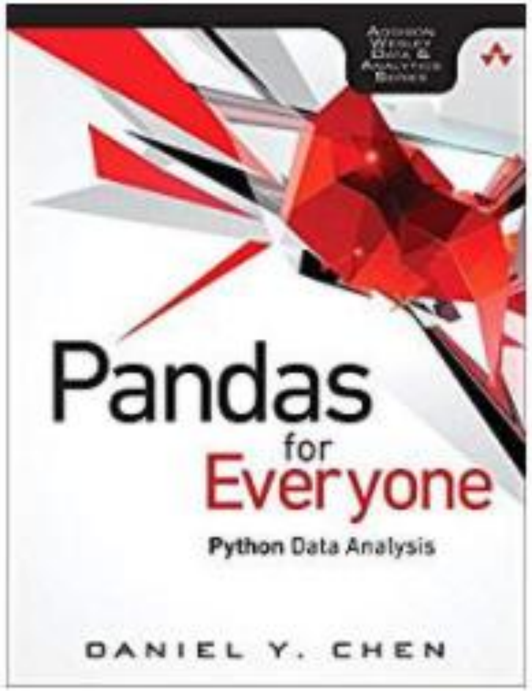
Kaggle 2017 DS & ML Survey

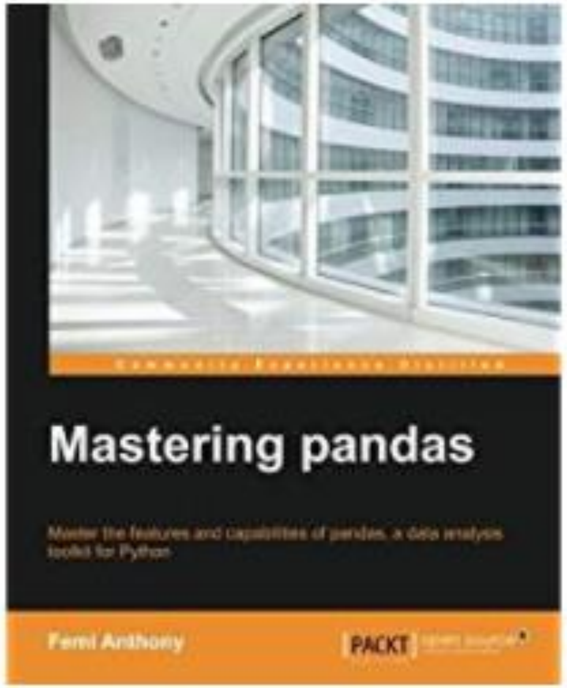


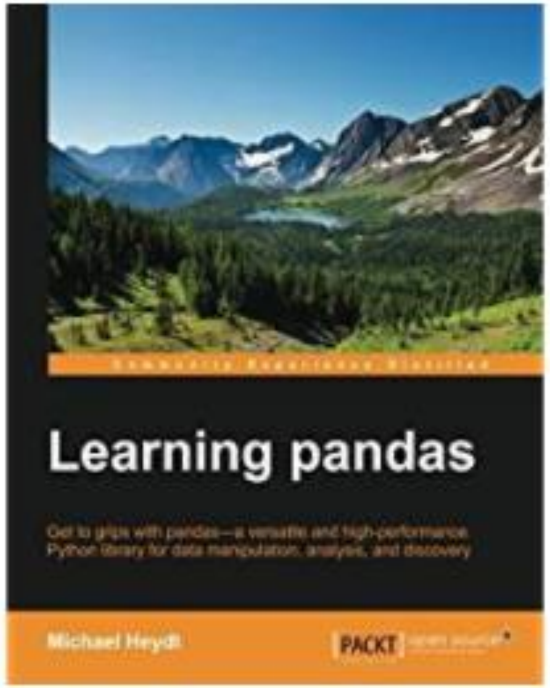
https://www.kaggle.com/surveys/2017

Many Pandas Books


TREADING ON PYTHON SERIES
Learning the
Pandas
Library
Python Tools for Data Munging, Data Analysis, and Visualization
Matt
Harrison




Community Experience Distilled
Mastering pandas
Master the features and capabilities of pandas, a data analysis toolkit for Python
Femi Anthony
PACKT open source


Community Experience Distilled
Learning pandas
Get to grips with pandas—a versatile and high-performance Python library for data manipulation, analysis, and discovery
Michael Heydt
PACKT open source

What is "Pandas" Module?

- panel data analysis or Python data analysis
- For building and manipulating "relational" or "tabular" data both easy and intuitive
- Built on top of NumPy (2005)
- Open source
  - Original author: Wes McKinney
  - Now part of the PyData project focused on improving Python data libraries
  - http://pandas.pydata.org

- >>> import panda as pd

Pandas History

- Developer Wes McKinney started working on Pandas in 2008 while at AQR Capital Management (global investment management firm)
- Need for a high performance, flexible analysis tool for quantitative analysis on financial data
- Before leaving AQR, he was able to convince management to allow him to open source the library
- Another AQR employee, Chang She, joined the effort in 2012 as the second major contributor to the library
- In 2015, Pandas signed on as a sponsored project of NumFOCUS, a non-profit charity in United States

Pandas Module

- Primary data structures
  - Series (1-dimensional)
  - DataFrame (2-dimensional) -- similar to *data.frame* in R
  - Panel (3-dimensional or more)
- Things that pandas does well
  - Easy handling of missing data
  - Size mutability: columns can be inserted and deleted (Add & drop columns)
  - Powerful, flexible group by functionality: Groupby & aggregation
  - Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
  - Intuitive merging and joining data sets: Join (merge) two data
  - Robust I/O tools for loading data from CSV & Excel files, database, and web sources

Pandas Series

Creating Pandas Series

- 1-D array of indexed data from Python list

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series([1,3,np.nan,6,8])
>>> s
0    1.0
1    3.0
2    NaN
3    6.0
4    8.0
dtype: float64
```

automatic indexing (record id/key)

- 1-D array of indexed data from NumPy ndarray

```
>>> a = np.array([1,3,np.nan,6,8])
>>> s2 = pd.Series(a)
>>> s2
0    1.0
1    3.0
2    NaN
3    6.0
4    8.0
dtype: float64
```

Creating Pandas Series (cont'd)

- 1-D array of indexed data from Python dictionary

```
>>> import pandas as pd
>>> import numpy as np
>>> d = {'spam':5.99, 'egg':0.99, 'ham':3.99}
>>> s = pd.Series(d)
egg     0.99
ham     3.99
spam    5.99
dtype: float64
```

pandas.Series()

- *pd*.Series([*data*], [*index*], [*dtype*], ...)
  - One-dimensional ndarray with axis labels (including time series)
  - *data*: Contains data stored in Series
  - *index*: Values must be hashable and have the same length as *data* (default: np.arange(len(*data*)))
  - Non-unique index values are allowed

```
>>> a = [2, 3, 5, 8]
>>> b = ['a', 'b', 'c', 'c']
>>> s = pd.Series(a)
>>> s
0    2
1    3
2    5
3    8
dtype: int64
>>> s2 = pd.Series(a, b)
>>> s2
a     2
b     3
c     5
c     8
dtype: int64
```

Handling Missing Entries

- Series creation from dictionary

```
>>> sdata = {'Ohio':35000,    \
             'Texas':71000,   \
             'Oregon':16000,  \
             'Utah':5000}
>>> s = pd.Series(sdata)
>>> s
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

- Extracting series-index from other list

```
>>> states = {'California',     \
              'Ohio', 'Oregon', \
              'Texas'}
>>> st = pd.Series(sdata, \
              index=states)
>>> st
Ohio          35000.0
Texas         71000.0
Oregon        16000.0
California        NaN
dtype: float64
```

value unknown

Checking Null Values

- *pd*.`isnull(`*obj*`)`
  - Return an array of Boolean indicating whether the corresponding element is missing
  - Same as *obj*.`isnull()`

- *pd*.`notnull(`*obj*`)`
  - Detect non-missing values
  - Same as *obj*.`notnull()`

```
>>> pd.isnull(st)
Ohio          False
Texas         False
Oregon        False
California     True
dtype: bool

>>> st.notnull()
Ohio           True
Texas          True
Oregon         True
California    False
dtype: bool
```

Pandas DataFrame

Creating Pandas DataFrame

- Dataframe is 2-D array of indexed data
  - Similar to a spreadsheet or SQL table
- Dataframe is the most commonly used pandas object
- Creating dataframe from Python dictionary

```
>>> import pandas as pd
>>> import numpy as np
>>> sales = {'Day':['M','T','W','Th','F'],\
        'Visitors':[43, 45, 33, 43, 78],  \
        'Revenue':[64, 73, 62, 64, 53]}
>>> df = pd.DataFrame(sales)
>>> df
```

automatic indexing (record id/key)

column

```
  Day  Revenue  Visitors
0   M       64        43
1   T       73        45
2   W       62        33
3  Th       64        43
4   F       53        78
```

pandas.DataFrame()

- *pd*.DataFrame([*data*], [*index*], [*columns*], [*dtype*], …)
  - The primary pandas data structure
  - Two-dimensional size-mutable, potentially heterogenous tabular data structure with labeled axes (rows and columns)
  - *data*: ndarray, list, dictionary, or dataframe
  - *index*: index to use for resulting frame. (default: `np.arange(len(`*data*`))`)
  - *columns*: column labels to use for resulting frame

```
>>> a = {'c0':[2, 3, 5, 8],     \
         'c1':[12, 76, 32, 29]}
>>> b = ['a', 'b', 'c', 'd']
>>> s = pd.DataFrame(a)
>>> s
   c0  c1
0   2  12
1   3  76
2   5  32
3   8  29
>>> s2 = pd.DataFrame(a, b)
>>> s2
   c0  c1
a   2  12
b   3  76
c   5  32
d   8  29
```

Indexing

- Row id = key = label = record id = index
- Used for
  - Accessing individual/multiple rows
  - Aligning multiple DataFrames and Series

- *df*.`set_index`(*keys*, ...)
  - Set the DataFrame index using existing columns
  - Return a new DataFrame with changed row labels (not in-place update)
  - *keys*: label or array/list of labels
    e.g., `df = df.set_index(['Day', 'Revenue'])`

```
>>> df.loc[2]
Day           W
Revenue      62
Visitors     33
Name: 2, dtype: object

>>> df = df.set_index('Day')
>>> df
     Revenue  Visitors
Day
M         64        43
T         73        45
W         62        33
Th        64        43
F         53        78
```

Accessing Rows/Columns

- Basic operations

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[0:2]`	DataFrame
Select rows by Boolean vector	`df[bool_vec]`	DataFrame

- `df.loc`: A slice object with labels [*start*:*stop*], both the *start* and the *stop* are included!

```
>>> df.loc['T']
Revenue     73
Visitors    45
Name: T, dtype: int64
>>> df.loc['M':'T'] inclusive!
     Revenue  Visitors
Day
M         64        43
T         73        45
>>> df.iloc[2:4]  exclusive!
     Revenue  Visitors
Day
W         62        33
Th        64        43
```

Accessing Rows/Columns (cont'd)

```
>>> df.loc[['M', 'F']]
     Revenue  Visitors
Day
M         64        43
F         53        78
>>> df['Visitors']
Day
M     43
T     45
W     33
Th    43
F     78
Name: Visitors, dtype: int64
```

```
>>> df[['Visitors','Revenue']]
     Visitors  Revenue
Day
M          43       64
T          45       73
W          33       62
Th         43       64
F          78       53
>>> df['Visitors']['M':'W']
Day
M    43
T    45
W    33
Name: Visitors, dtype: int64
```

Boolean Indexing

```
>>> df[[True,False,False,True,False]]
     Revenue  Visitors
Day
M         64        43
Th        64        43
>>> df[df['Revenue'] > 65]
     Revenue  Visitors
Day
T         73        45
>>> df[(df['Revenue'] > 50) &  \
       (df['Visitors'] > 50)]
     Revenue  Visitors
Day
F         53        78
```

```
>>> df[(df['Revenue'] > 50) |  \
       (df['Visitors'] > 50)]
     Revenue  Visitors
Day
T         73        45
F         53        78
>>> df[df > 50]
     Revenue  Visitors
Day
M         64       NaN
T         73       NaN
W         62       NaN
Th        64       NaN
F         53      78.0
```

Column Manipulation

- Change the order of columns

```
>>> df2 = pd.DataFrame(df, columns=['Visitors','Revenue'])
>>> df2
     Visitors  Revenue
Day
M          43       64
T          45       73
W          33       62
Th         43       64
F          78       53
```

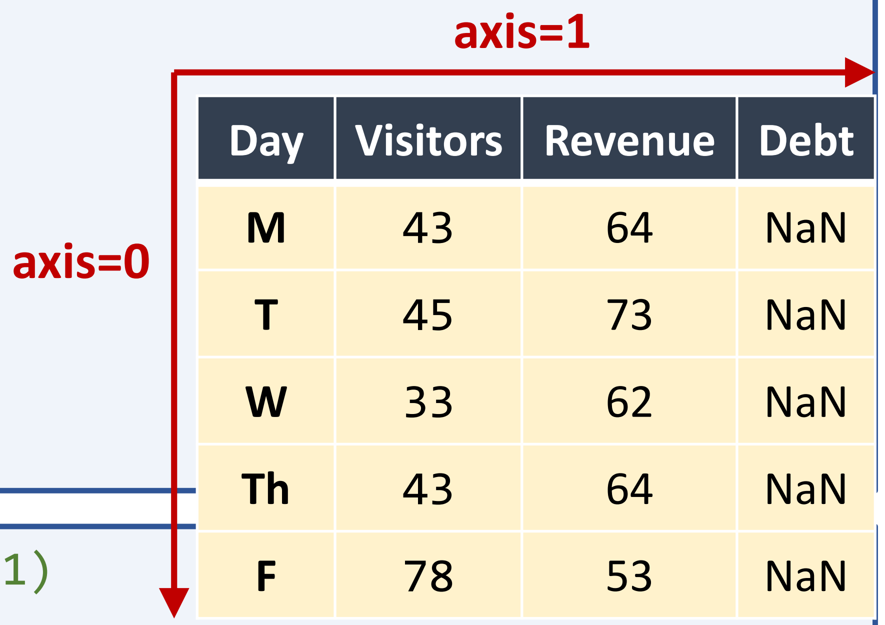
- Add a new column:
  - NaN are filled to added column values

```
>>> df3 = pd.DataFrame(df, columns=['Visitors','Revenue','Debt'])
>>> df3
     Visitors  Revenue  Debt
Day
M          43       64   NaN
T          45       73   NaN
W          33       62   NaN
Th         43       64   NaN
F          78       53   NaN
```

Column Manipulation (cont'd)

- Delete an existing column
  - Using `del` (delete in-place)

```
>>> del df3['Debt']
>>> df3
     Visitors  Revenue
Day
M          43       64
T          45       73
W          33       62
Th         43       64
F          78       53
```



Day	Visitors	Revenue	Debt
M	43	64	NaN
T	45	73	NaN
W	33	62	NaN
Th	43	64	NaN
F	78	53	NaN

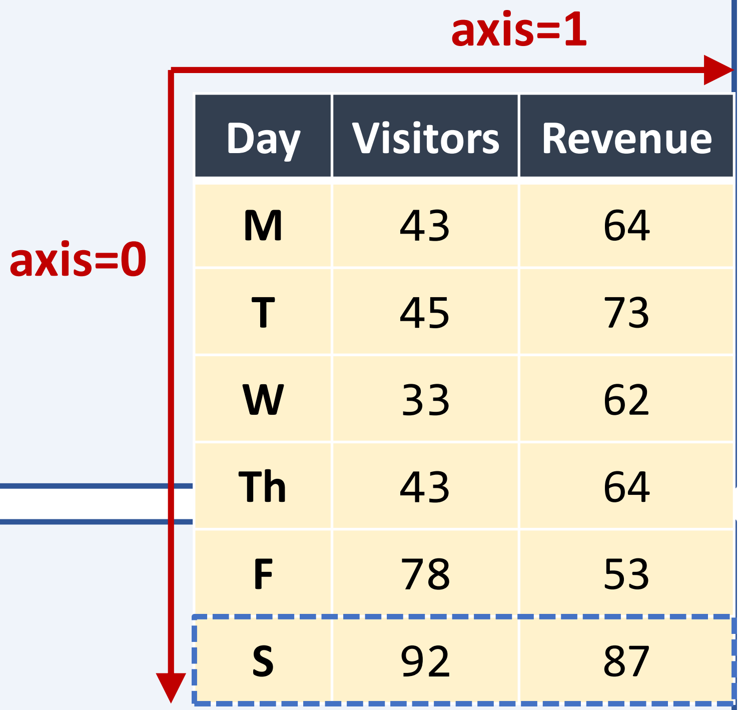
  - Using `df.drop()` (return a new df)

```
>>> df3.drop('Debt', axis=1)
     Visitors  Revenue
Day
M          43       64
T          45       73
W          33       62
Th         43       64
F          78       53
```

Row Manipulation

- Add a new row

```
>>> df2.loc['S'] = [92, 87]
>>> df2
 Visitors  Revenue
Day
M           43       64
T           45       73
W           33       62
Th          43       64
F           78       53
S           92       87
```



Day	Visitors	Revenue
M	43	64
T	45	73
W	33	62
Th	43	64
F	78	53
S	92	87

- Delete an existing row
  - Using `df.drop()` (return a new df)

```
>>> df2.drop('S', axis=0)
Visitors  Revenue  Debt
Day
M           43       64
T           45       73
W           33       62
Th          43       64
F           78       53
```

Rename Row/Column

- *df*.rename([*index*], [*columns*], [*inplace*], ...)
  - Rename any index, row or column
  - A part of rows or columns can be altered
  - *index*: dict. for changing row indexes
  - *columns*: dict. for changing column indexes
  - *inplace*: If True, *df* is updated in place. Otherwise, return a new *df* (default: False)

```
>>> newr = {'M':'Mo', 'T':'Tu'}
>>> df.rename(index=newr)
     Revenue  Visitors
Day
Mo        64        43
Tu        73        45
W         62        33
Th        64        43
F         53        78
>>> newc = {'Revenue':'Rev.'}
>>> df.rename(columns=newc)
     Rev.  Visitors
Day
M      64        43
T      73        45
W      62        33
Th     64        43
F      53        78
```

Common Statistical Functions

Method	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of values
median()	Arithmetic median of values
min()	Minimum
max()	Maximum
std()	Bessel-corrected sample standard deviation
var()	Unbiased variance
skew()	Sample skewness (3rd moment)
kurt()	Sample kurtosis (4th moment)
quantile()	Sample quantile (value at %)
apply()	Generic apply
cov()	Unbiased covariance
corr()	Correlation

- Applicable both Series and DataFrame objects

```
>>> df[df['Revenue']>60].count()
Revenue     4
Visitors    4
dtype: int64
>>> df['Revenue'].mean()
63.2
>>> df.cov()
          Revenue  Visitors
Revenue     50.70    -81.35
Visitors   -81.35    295.80
>>> df.corr()
           Revenue  Visitors
Revenue   1.000000 -0.664285
Visitors -0.664285  1.000000
```

Iteration over Rows

- *df*.`iterrows()`
  - Iterate over rows of DataFrame as (index, Series) pairs

```
>>> df = pd.DataFrame(data=[[1,2,3],[4,5,6],[7,8,9]], columns=['A','B','C'])
>>> df
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9

>>> for index, row in df.iterrows():
...     print(row['A'], row['B'])
1 2
4 5
7 8
```

```
>>> for index, row in df.iterrows():
...     print(row)
A    1
B    2
C    3
Name: 0, dtype: int64
A    4
B    5
C    6
Name: 1, dtype: int64
A    7
B    8
C    9
Name: 2, dtype: int64
```

Series

Groupby and Aggregation (1)

- Example DataFrame object
  - 2 categorical values: A, B
  - 2 numerical values: C, D

```
>>> 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)})
>>> df
     A      B         C         D
0  foo    one -0.578235  0.193109
1  bar    one  1.312911  0.576292
2  foo    two  0.628944  0.484595
3  bar  three  0.206827  0.810682
4  foo    two  1.584507  1.153200
5  bar    two -0.367555 -0.703818
6  foo    one -0.017915 -1.297967
7  foo  three  0.337489  1.565021
```

```
>>> df[df['A'] == 'foo']
     A      B         C         D
0  foo    one -0.578235  0.193109
2  foo    two  0.628944  0.484595
4  foo    two  1.584507  1.153200
6  foo    one -0.017915 -1.297967
7  foo  three  0.337489  1.565021
>>> df[df['A'] == 'bar']
     A      B         C         D
1  bar    one  1.312911  0.576292
3  bar  three  0.206827  0.810682
5  bar    two -0.367555 -0.703818
```

Groupby and Aggregation (2)

- *df*.groupby([*by*], [*axis*], ...)
  - Used to group large amounts of data and compute operations on these groups
  - *by*: label, function, a list of labels, ... (Used to determine the groups)
  - *axis*: 0 or '*index*' for rows, 1 or '*columns*' for columns (default: 0)

- Aggregation stat functions after grouping
  - `mean()`, `sum()`, `median()`, `var()`, etc.

```
>>> g = df.groupby('A')
>>> g.mean()
            C         D
A
bar  0.384061  0.227719
foo  0.390958  0.419592
>>> g.sum()
            C         D
A
bar  1.152183  0.683156
foo  1.954789  2.097958
>>> g.corr()
              C         D
A
bar C  1.000000  0.661077
    D  0.661077  1.000000
foo C  1.000000  0.493865
    D  0.493865  1.000000
```

Groupby and Aggregation (3)

- Get a group's contents

```
>>> g.get_group('bar')
       B         C         D
1    one  1.312911  0.576292
3  three  0.206827  0.810682
5    two -0.367555 -0.703818
```

- Printing the groups

```
>>> for key, item in g:
...     print(key)
...     print(g.get_group(key))
bar
       B         C         D
1    one  1.312911  0.576292
3  three  0.206827  0.810682
5    two -0.367555 -0.703818
foo
       B         C         D
0    one -0.578235  0.193109
2    two  0.628944  0.484595
4    two  1.584507  1.153200
6    one -0.017915 -1.297967
7  three  0.337489  1.565021
```

Groupby and Aggregation (4)

- Describing a group

```
>>> g.describe()
 C                                                              \
    count      mean       std       min       25%       50%       75%
A
bar   3.0  0.384061  0.854137 -0.367555 -0.080364  0.206827  0.759869
foo   5.0  0.390958  0.804762 -0.578235 -0.017915  0.337489  0.628944

                  D
\
          max count      mean       std       min       25%       50%
A
bar  1.312911   3.0  0.227719  0.815202 -0.703818 -0.063763  0.576292
foo  1.584507   5.0  0.419592  1.101784 -1.297967  0.193109  0.484595


          75%       max
A
bar  0.693487  0.810682
foo  1.153200  1.565021
```

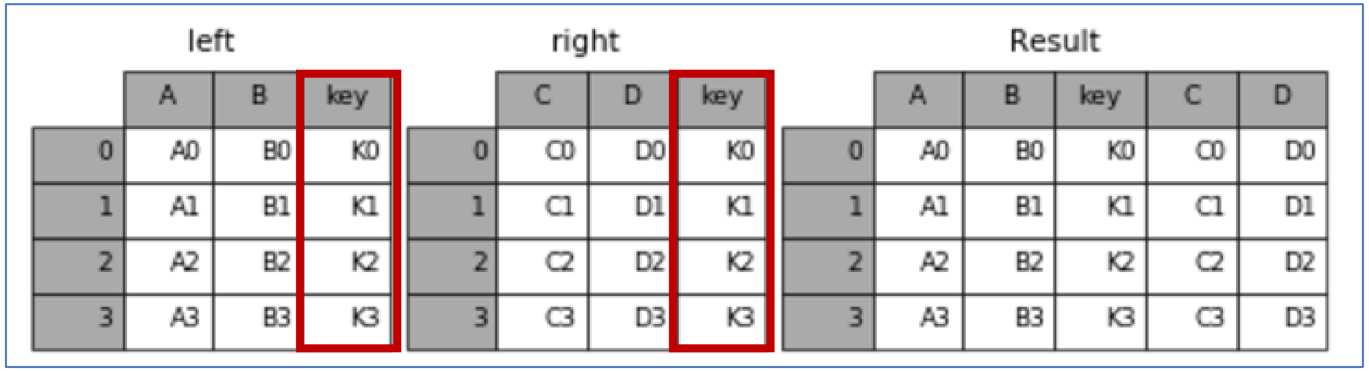
- Grouping by multiple columns

```
>>> gm = df.groupby(['A','B'])
>>> gm.mean()
                  C         D
A   B
bar one    1.312911  0.576292
    three  0.206827  0.810682
    two   -0.367555 -0.703818
foo one   -0.298075 -0.552429
    three  0.337489  1.565021
    two    1.106725  0.818898
```

Merging (Joining)

- *pd*.merge(*left*, *right*, [*how*], [*on*], [*left_on*], [*right_on*], [*left_index*], [*right_index*], …)
  - Merge DataFrame objects with database-style join
  - *left*: DataFrame
  - *right*: Object to merge with
  - *how*: join type -- 'left', 'right', 'outer', or 'inner' (default: 'inner')
  - *on*: column to join on (label or list) -- must be found on both DataFrames
  - *left_on* (or *right_on*): column to join on in the left (or right) DataFrame
  - *left_index* (or *right_index*): if True, use the index from the left (or right) DataFrame

```
result =
  pd.merge(left, right, on='key')
```



left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	B3	K3

right

	C	D	key
0	C0	D0	K0
1	C1	D1	K1
2	C2	D2	K2
3	C3	D3	K3

Result

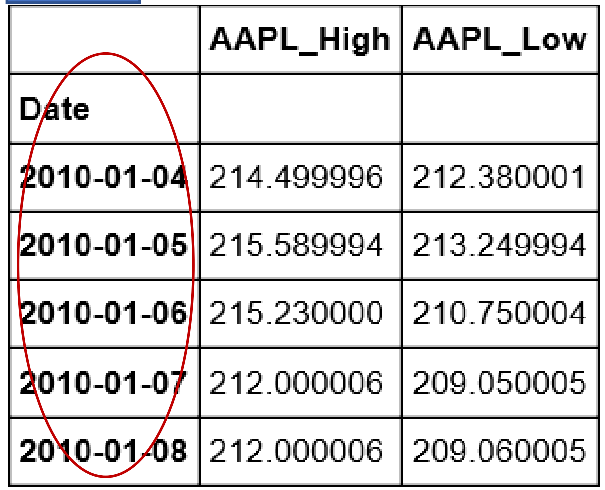
	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K2	C2	D2
3	A3	B3	K3	C3	D3

Merging (Joining) (cont'd)

- Merge two DataFrames by their own index

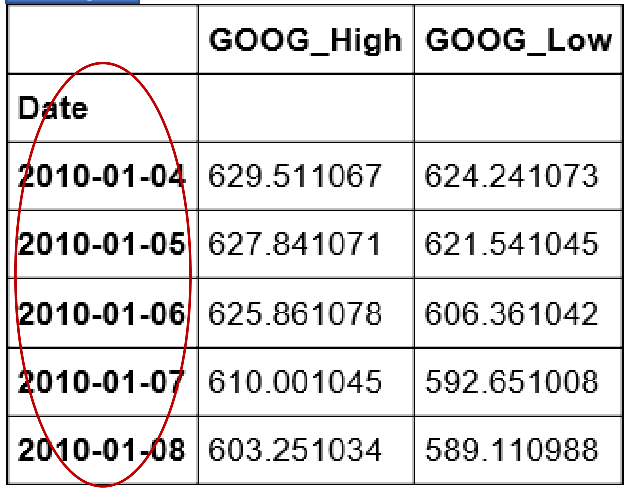
```
dfxy = pd.merge(dfx, dfy, left_index=True, right_index=True)
```

dfx



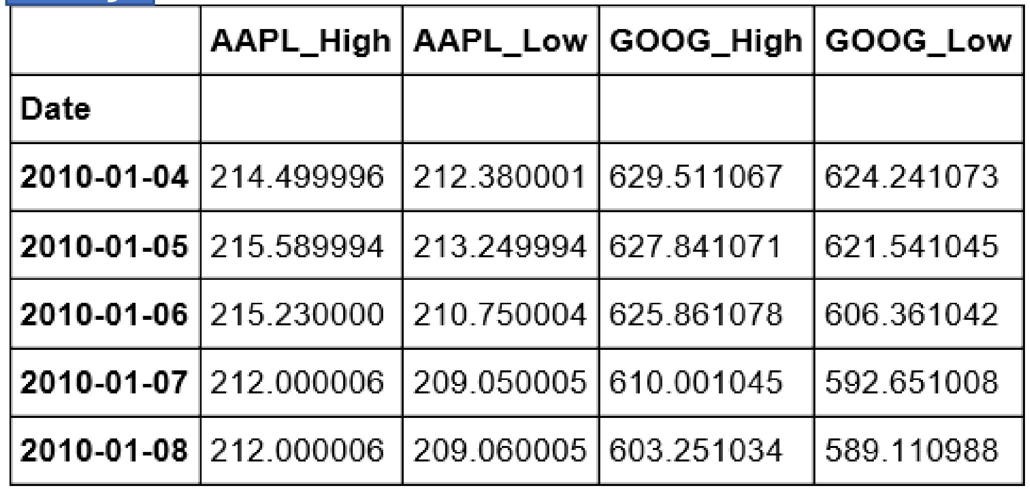
	AAPL_High	AAPL_Low
Date		
2010-01-04	214.499996	212.380001
2010-01-05	215.589994	213.249994
2010-01-06	215.230000	210.750004
2010-01-07	212.000006	209.050005
2010-01-08	212.000006	209.060005

dfy



	GOOG_High	GOOG_Low
Date		
2010-01-04	629.511067	624.241073
2010-01-05	627.841071	621.541045
2010-01-06	625.861078	606.361042
2010-01-07	610.001045	592.651008
2010-01-08	603.251034	589.110988

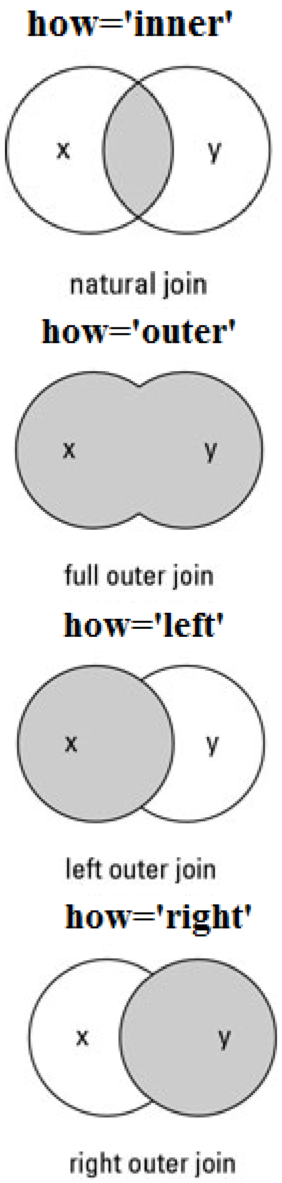
dfxy



	AAPL_High	AAPL_Low	GOOG_High	GOOG_Low
Date				
2010-01-04	214.499996	212.380001	629.511067	624.241073
2010-01-05	215.589994	213.249994	627.841071	621.541045
2010-01-06	215.230000	210.750004	625.861078	606.361042
2010-01-07	212.000006	209.050005	610.001045	592.651008
2010-01-08	212.000006	209.060005	603.251034	589.110988

Merging (Joining) Types

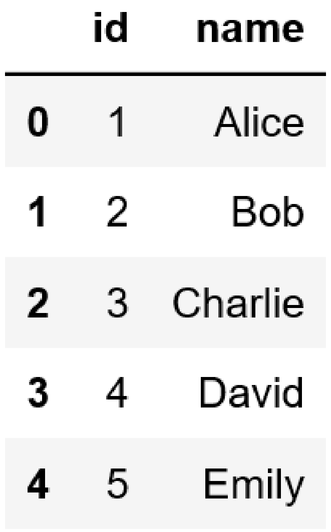
- Inner join ('inner')
  - Return only the rows in which the left table have matching keys in the right table
- Outer join ('outer')
  - Returns all rows from both tables, join records from the left which have matching keys in the right table.
- Left outer join ('left')
  - Return all rows from the left table, and any rows with matching keys from the right table.
- Right outer join ('right')
  - Return all rows from the right table, and any rows with matching keys from the left table.



Merging (Joining) Example

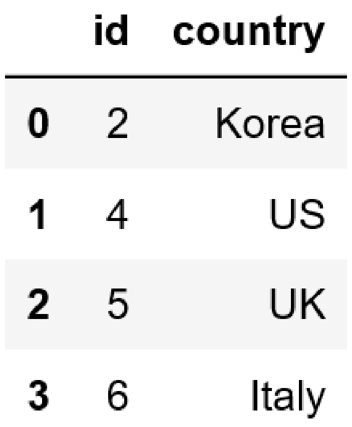
`pd.merge(df1, df2)`

df1



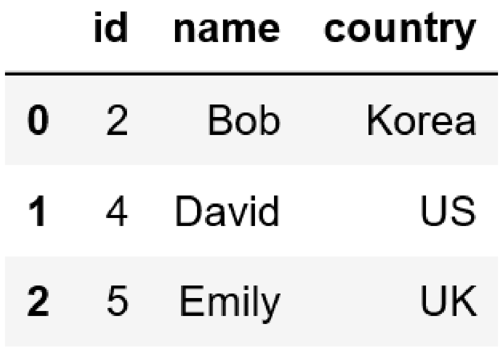
	id	name
0	1	Alice
1	2	Bob
2	3	Charlie
3	4	David
4	5	Emily

df2



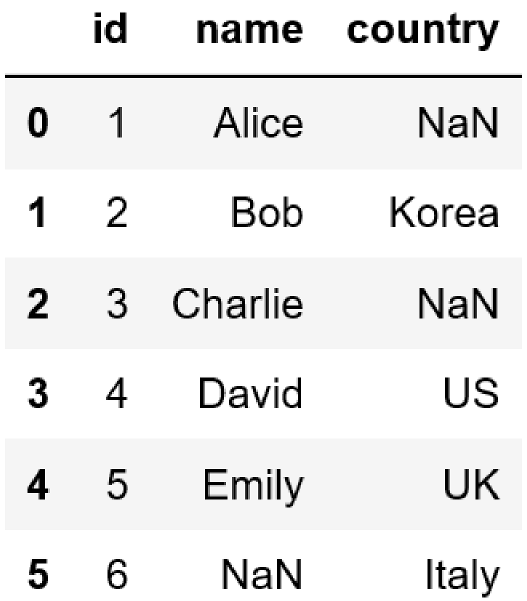
	id	country
0	2	Korea
1	4	US
2	5	UK
3	6	Italy

inner



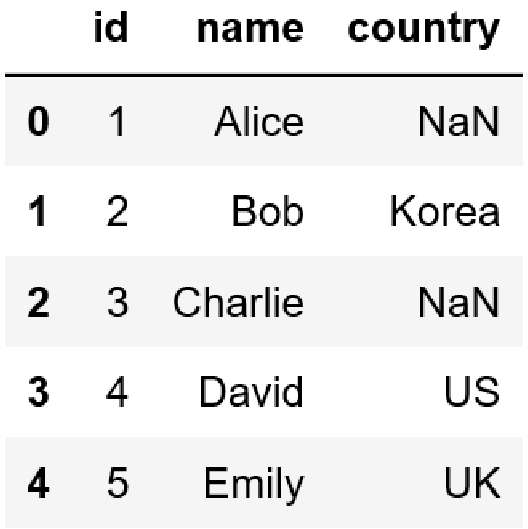
	id	name	country
0	2	Bob	Korea
1	4	David	US
2	5	Emily	UK

outer



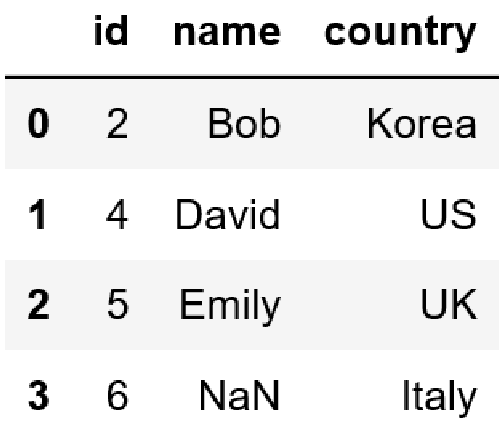
	id	name	country
0	1	Alice	NaN
1	2	Bob	Korea
2	3	Charlie	NaN
3	4	David	US
4	5	Emily	UK
5	6	NaN	Italy

left



	id	name	country
0	1	Alice	NaN
1	2	Bob	Korea
2	3	Charlie	NaN
3	4	David	US
4	5	Emily	UK

right



	id	name	country
0	2	Bob	Korea
1	4	David	US
2	5	Emily	UK
3	6	NaN	Italy

I/O in Pandas

I/Os for Pandas DataFrame

- A collection of convenient I/O functions supporting various file formats

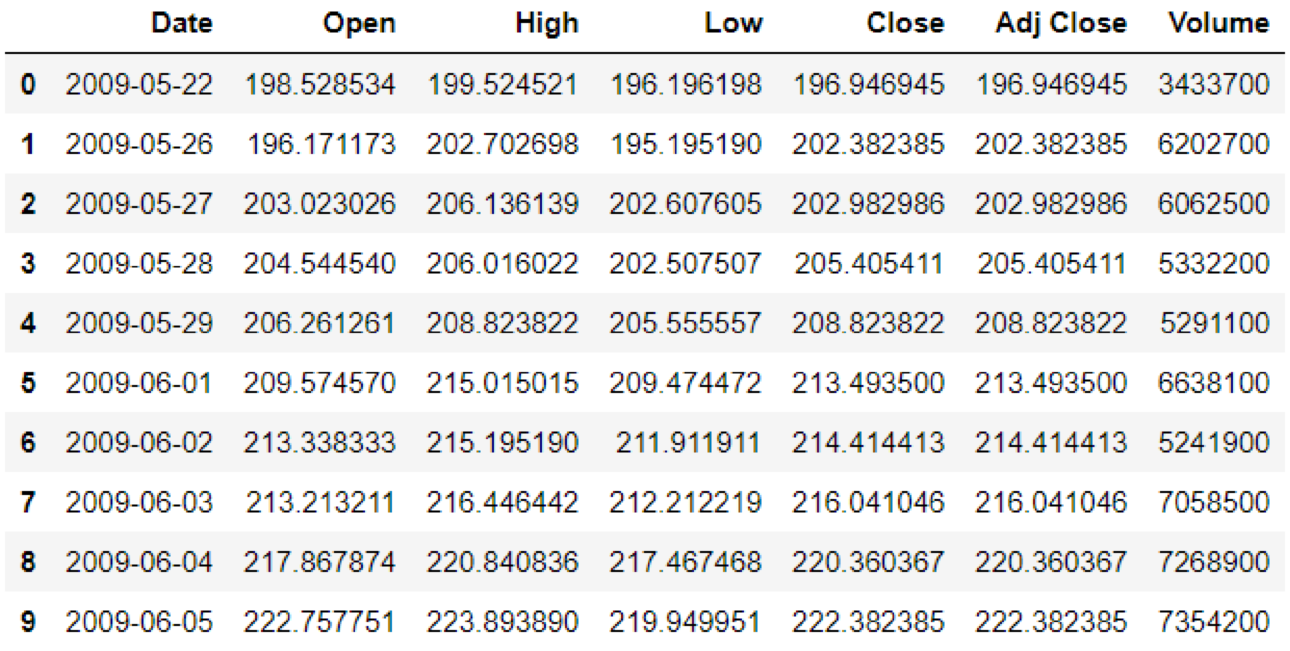
```
to_csv()     to_excel()     to_hdf()     to_sql()     to_json()     to_html()
read_csv()   read_excel()   read_hdf()   read_sql()   read_json()   read_html()
```

  - (cf.) HDF (Hierarchical Data Format): Standardized file format for scientific data

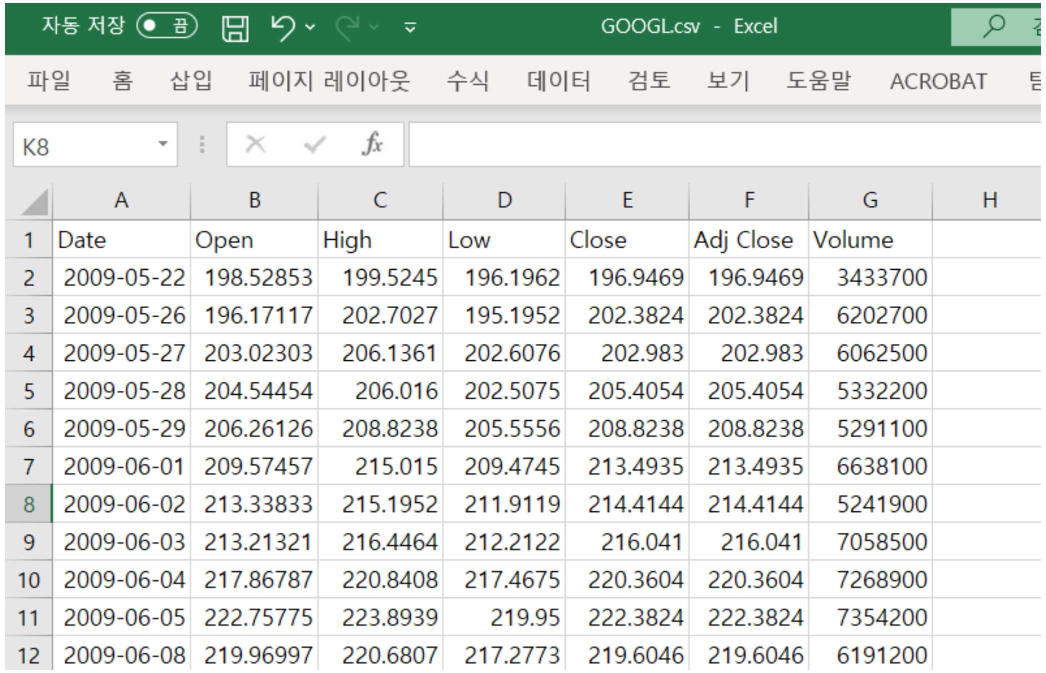
- From Pandas DataFrame to CSV file: *pd*.to_csv(*path*)
- From CSV file to Pandas DataFrame: *pd*.read_csv(*path*)

Reading a CSV File

```
>>> df = pd.read_csv('GOOGL.csv')
>>> df.iloc[0:9]
```



```
         Date        Open        High         Low       Close   Adj Close   Volume
0  2009-05-22  198.528534  199.524521  196.196198  196.946945  196.946945  3433700
1  2009-05-26  196.171173  202.702698  195.195190  202.382385  202.382385  6202700
2  2009-05-27  203.023026  206.136139  202.607605  202.982986  202.982986  6062500
3  2009-05-28  204.544540  206.016022  202.507507  205.405411  205.405411  5332200
4  2009-05-29  206.261261  208.823822  205.555557  208.823822  208.823822  5291100
5  2009-06-01  209.574570  215.015015  209.474472  213.493500  213.493500  6638100
6  2009-06-02  213.338333  215.195190  211.911911  214.414413  214.414413  5241900
7  2009-06-03  213.213211  216.446442  212.212219  216.041046  216.041046  7058500
8  2009-06-04  217.867874  220.840836  217.467468  220.360367  220.360367  7268900
9  2009-06-05  222.757751  223.893890  219.949951  222.382385  222.382385  7354200
```



	A	B	C	D	E	F	G	H
1	Date	Open	High	Low	Close	Adj Close	Volume	
2	2009-05-22	198.52853	199.5245	196.1962	196.9469	196.9469	3433700	
3	2009-05-26	196.17117	202.7027	195.1952	202.3824	202.3824	6202700	
4	2009-05-27	203.02303	206.1361	202.6076	202.983	202.983	6062500	
5	2009-05-28	204.54454	206.016	202.5075	205.4054	205.4054	5332200	
6	2009-05-29	206.26126	208.8238	205.5556	208.8238	208.8238	5291100	
7	2009-06-01	209.57457	215.015	209.4745	213.4935	213.4935	6638100	
8	2009-06-02	213.33833	215.1952	211.9119	214.4144	214.4144	5241900	
9	2009-06-03	213.21321	216.4464	212.2122	216.041	216.041	7058500	
10	2009-06-04	217.86787	220.8408	217.4675	220.3604	220.3604	7268900	
11	2009-06-05	222.75775	223.8939	219.95	222.3824	222.3824	7354200	
12	2009-06-08	219.96997	220.6807	217.2773	219.6046	219.6046	6191200	

pandas.read_csv()

- *pd*.read_csv(*filepath*, [*sep*], [*header*], [*names*], [*index_col*], [*encoding*], ...)
  - Read a comma-separated values (csv) file
  - *filepath*: any valid string path. The string could be a URL.
  - *sep* (or *delimiter*): delimiter to use (default: ' , ')
  - *header*: row number(s) to use as the column names
  - *names*: list of column names to use
  - *index_col*: column(s) to use as the row labels of the Data Frame
  - *encoding*: encoding to use (default: 'utf-8')

```
>>> df2 = pd.read_csv('GOOGL.csv', names=['A','B','C','D','E','F'])
>>> df2 = pd.read_csv('mydf.csv', sep=':')
```

DataFrame.to_csv()

- *df*.to_csv(*filepath*, [*sep*], [*columns*], [*header*], [*index*], [*encoding*], ...)
  - Write DataFrame to a comma-separated values (csv) file
  - *filepath*: any valid string path.
  - *sep* (or *delimiter*): delimiter to use (default: ' , ')
  - *columns*: columns to write
  - *header*: write out the column names (default: True)
  - *index*: write row names (default: True)
  - *encoding*: encoding to use (default: 'utf-8')

```
>>> df.to_csv('mydf.csv', sep='\t')
>>> df.to_csv('dataset.csv', sep='\t', encoding='utf-8')
```