Analyzing and Manipulating data with

pandas

Pandas data I/O

- **Pandas** provides a high-level interface to and from many file formats used in **data science**:
  - .txt, csv, json, HTML, Excel(.xls .xlsx), pickle, HDF5, SQL, R, …
- For any given format, there is
  - A read_** function,
  - A to_** method attached to all Pandas data objects

Pandas' read_table **example**

- Features
  - Read_table can read tabular text (CSV files) into a **DataFrame** and implements the following
    - Detect comments, headers and footers
    - Specify which column is the index
    - Specify the column name or which line is the column name
    - Parse dates stored in 1 column or in multiple
    - Manage multiple codes for missing data
    - Read data by chunk (large files)
    - Custom conversion of values based on column

Pandas' read_table example

- Example

Historical_data.csv

```
Date,Open,High,Low,Close,Adj Close,Volume
2018-02-28,27.650000,27.719999,27.209999,27.219999,,61700
2018-03-01,27.150000,27.639999,27.139999,27.559999,27.559999,184300
2018-03-02,27.410000,28.150000,27.410000,27.850000,27.850000,149200
2018-03-05,27.850000,28.040001,27.510000,27.690001,27.690001,90400
2018-03-06,27.670000,28.129999,27.450001,28.049999,28.049999,88700
```

In [1]:
```python
import pandas as pd
```

In [2]:
```python
data = pd.read_table('D:\Tableau\Historical_data.csv', sep=',', header=0, na_values=['-'],parse_dates=True)
```

In [3]:
```python
data.head()
```

Out[3]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2018-02-28	27.65	27.719999	27.209999	27.219999	NaN	61700
1	2018-03-01	27.15	27.639999	27.139999	27.559999	27.559999	184300
2	2018-03-02	27.41	28.150000	27.410000	27.850000	27.850000	149200
3	2018-03-05	27.85	28.040001	27.510000	27.690001	27.690001	90400
4	2018-03-06	27.67	28.129999	27.450001	28.049999	28.049999	88700

Pandas IO summary

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Storing data in **Pandas**

Pandas data structures

- PANDA = **PAN**el **DAt**a **S** = multi-dimensional data in stats & econometrics. Introduces 3 size-mutable, labeled data-structures:
  - A **Series** is a 1D data-structure.
  - A **DataFrame** is a 2D data-structure that can be viewed as a dictionary of Series.
  - A **Panel** is a 3D data-structure that can be viewed as a dictionary of DataFrames

Series

Conceptually, **pandas.Series** are indexed arrays:

- NumPy arrays map a range of integers to values
- Series map arbitrary sets of labels to values
- Series may also be seen as a specialized, ordered dictionary where values all have the same type and are stored efficiently

In [1]:
```
import pandas as pd
```

In [2]:
```
s = pd.Series({'a':0,'b':1,'c':2,'d':3})
```

#Dict-like (key-value) access can be label-based

In [3]:
```
s['b']
```

Out[3]:
```
1
```

#The labels are accessed via the s.index attribute and the values by the s.values attribute (NumPy array).

Creating Series

FROM LIST AND DICT

In [1]:
```
from pandas import Series
```

#Data and corresponding indices can

#be stored in lists.

In [2]:
```
index = ['a', 'b', 'c', 'd']
Series(range(4), index=index,name='first series')
```

Out[2]:
```
a    0
b    1
c    2
d    3
Name: first series, dtype: int64
```

#data + indices in a dict

In [3]:
```
d = {'a':0,'b':1,'c':2,'d':3}
s = Series(d, name='first series')
s.index
```

Out[3]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [4]:
```
s.values, type(s.values)
```

Out[4]:
```
(array([0, 1, 2, 3], dtype=int64), numpy.ndarray)
```

In [5]:
```
s.dtype
```

Out[5]:
```
dtype('int64')
```

ACCESS OR ADD ELEMENTS

#Request existing values

In [6]:
```
s['b']
```

Out[6]:
```
1
```

#Modify and existing value

In [7]:
```
s['b']=3
```

#Add new elements

In [9]:
```
s['e']=5
```

In [10]:
```
s
```

Out[10]:
```
a    0
b    3
c    2
d    3
e    5
Name: first series, dtype: int64
```

DataFrames

A **DataFrame** object can be viewed as a dictionary of Series sharing a common index:

- Dataframes have both row (index) and column (columns) indices
- Each column may have a different type
- Adding a column is 'cheap'

In [1]:
```
from pandas import Series, DataFrame
```

In [2]:
```
s1 = Series({'a': 1, 'b': 2, 'c': 3})
```

In [3]:
```
s2 = Series({'a': True, 'b': False, 'c': True})
```

In [4]:
```
df = DataFrame({'col1': s1, 'col2': s2})
```

#Dict-like access is column-based

In [5]:
```
df['col1']
```

Out[5]:
```
a    1
b    2
c    3
Name: col1, dtype: int64
```

Creating DataFrames

```
In [1]: from pandas import Series, DataFrame

In [2]: index=['a', 'b', 'c', 'd']

In [3]: s1 = Series({'a':0, 'b':1, 'c':2, 'd':3})

In [4]: s2=Series([-0.9, -1.7, 1.1], index=index[1:])

In [5]: d = {'A':s1, 'B':s2}

In [6]: df = DataFrame(d)
        print(df)

   A    B
a  0  NaN
b  1 -0.9
c  2 -1.7
d  3  1.1
```

```
In [7]: print(df.index)
        print(df.columns)

Index(['a', 'b', 'c', 'd'], dtype='object')
Index(['A', 'B'], dtype='object')

In [8]: print('dimension of dataframe:',df.shape)
        print('data types in dataframe\n',df.dtypes)

dimension of dataframe: (4, 2)
data types in dataframe
 A      int64
B    float64
dtype: object

In [9]: print(df.values)

[[ 0.   nan]
 [ 1.  -0.9]
 [ 2.  -1.7]
 [ 3.   1.1]]
```

Accessing values in Pandas

Pandas and indexing

ESSentially:

Series[label] -> scalar
DataFrame[label] -> column

```
In [1]: from pandas import Series, DataFrame

In [2]: s=Series({'a':0, 'b':1, 'c':2})

In [3]: s['a']
Out[3]: 0

In [4]: df = DataFrame({'A':s , 'B': -s})

In [5]: df['A']
Out[5]: a    0
        b    1
        c    2
        Name: A, dtype: int64

In [6]: df['B']
Out[6]: a    0
        b   -1
        c   -2
        Name: B, dtype: int64
```

Series and DataFrames have powerful indexing capabilities:

- Values are accessible as NumPy arrays
- More interestingly: label-based indexing
- Indices allow **automatic** alignment: especially interesting with timeseries, and for NaN (missing data) handling

BUT if you do slicing

```
In [7]: df[:2] # first two rows !!
Out[7]:
```

	A	B
a	0	0
b	1	-1

Pandas and indexing

Label-based vs position-based indexing

- Indexing operator [] has an ambiguity:
  - Series[integer_value]: position or label?
  - DataFrame[integer_value]: position or column name ?

- .loc attribute: purely “label”
- .iloc attribute: purely index-based, aka position (integer value)

```
In [1]: from pandas import Series, DataFrame

In [2]: s=Series({'a':0, 'b':1, 'c':2})

In [3]: s.iloc[1]
Out[3]: 1

In [4]: s.iloc['a']

In [5]: s.loc['a']
Out[5]: 0

In [6]: s.loc[0]
```

Indexing into **Series**

ACCESING 1 ELEMENT

In [1]:
```
from pandas import Series, DataFrame
```

In [2]:
```
s=Series({'a':0, 'b':1, 'c':2, 'd':3})
```

In [3]:
```
# Access elements based on position
s.iloc[2]
```

Out[3]: 2

In [4]:
```
# Access elements based on label
s.loc['c']
```

Out[4]: 2

In [5]:
```
# indexing into a Series is equivalent
s['c']
```

Out[5]: 2

SLICING ELEMENTS OUT

In [6]:
```
# From:  s.iloc[pos_lower: pos_upper: step]
s.iloc[:2]
```

Out[6]:
```
a    0
b    1
dtype: int64
```

In [7]:
```
# Every other element
s.iloc[::2]
```

Out[7]:
```
a    0
c    2
dtype: int64
```

In [8]:
```
# From:  s.loc[label_lower: label_upper: step]
s.loc['a':'c']
```

Out[8]:
```
a    0
b    1
c    2
dtype: int64
```

FANCY-INDEXING

In [9]:
```
# Custom selection of elements
s[[True, False, True, True]]
```

Out[9]:
```
a    0
c    2
d    3
dtype: int64
```

In [10]:
```
# Masks can be created by comparing
# Values in the series or another on
s>1
```

Out[10]:
```
a    False
b    False
c     True
d     True
dtype: bool
```

In [11]:
```
s[s>1]
```

Out[11]:
```
c    2
d    3
dtype: int64
```

Indexing into **DataFrames**

ACCESS ELEMENTS

In [1]:
```
from pandas import Series, DataFrame
```

In [2]:
```
index=['a', 'b', 'c', 'd']

s1 = Series({'a':0, 'b':1, 'c':2, 'd':3})

s2=Series([-0.9, -1.7, 1.1], index=index[1:])

d = {'A':s1, 'B':s2}
```

In [3]:
```
df = DataFrame(d)
df.head()
```

Out[3]:

	A	B
a	0	NaN
b	1	-0.9
c	2	-1.7
d	3	1.1

In [4]:
```
# 1 (or more) column accessed like a dict..
df['A']
```

Out[4]:
```
a    0
b    1
c    2
d    3
Name: A, dtype: int64
```

In [5]:
```
# or like an object
series2 = df.B
#Access all columns for 1 index
df.loc['c']
```

Out[5]:
```
A    2.0
B   -1.7
Name: c, dtype: float64
```

In [6]:
```
# or 1 element of the table
df.loc['c','B']
```

Out[6]: -1.7

SLICING ELEMENTS OUT

Form df.loc[row lower: row upper : step, col lower : col upper : step]

In [8]:
```
sub_df = df.loc['c':, 'A':'B']
```

In [9]:
```
# Incomplete slicing assumes all
# elements in other dimesions
df.loc['c':]
```

Out[9]:

	A	B
c	2	-1.7
d	3	1.1

MIXED INDEXING

Mixed indexing using .ix:

In [13]:
```
sub_df = df.ix[2,'B']
sub_df
```

Out[13]: -1.7

TASK01

- Create a DataFrame with random data:

```
In [1]: import pandas as pd
        import numpy as np
        data = np.arange(12).reshape(4, 3)
        print(data)

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

In [2]: df = pd.DataFrame( data, index = ['one', 'two', 'three', 'four'], columns=['X', 'Y', 'Z'])
```

1. Get column 'Y'
2. Get row 'three' (by name)
3. Get the second and fourth row (by index)
4. Get the columns 'Y' and 'Z' of rows 'two' and 'three'

Re-indexing

- The index of a Pandas data-structure is the key that controls:
  - how the data is displayed and ordered,
  - how to align and combine different datasets.

- The index can be:
  - shuffled (and the values will follow),
  - overwritten,
  - transformed,
  - set to the values of any of the columns of a **DataFrame** ,
  - made of multiple sub-indices.

Re-indexing **Series**

ALIGNMENT OF 2 SERIES

```
In [5]: s = Series(range(4), index=index)
        s2 = s.iloc[:-2]
        s2

Out[5]: a    0
        b    1
        dtype: int64

In [6]: # Operations automatically align on
        # the index (different from NumPy)
        s + s2

Out[6]: a    0.0
        b    2.0
        c    NaN
        d    NaN
        dtype: float64
```

RE-INDEXING

```
In [1]: from pandas import Series, DataFrame

In [2]: index = ['a', 'b', 'c', 'd']
        s = Series(range(4), index=index)
        print(s)

        a    0
        b    1
        c    2
        d    3
        dtype: int64

In [3]: # Select a different set of indices
        s=s.reindex(['c', 'b', 'a', 'e'])
        print(s)

        c    2.0
        b    1.0
        a    0.0
        e    NaN
        dtype: float64

In [4]: # Sort by values. See s.sort_value()
        # to sort based on index value.
        s.sort_index(ascending=False)

Out[4]: e    NaN
        c    2.0
        b    1.0
        a    0.0
        dtype: float64
```

Re-indexing DataFrames

RE-INDEXING DATAFRAMES

In [1]:
```
from pandas import Series, DataFrame
```

In [2]:
```
index=['a', 'b', 'c', 'd']
s1 = Series({'a':0, 'b':1, 'c':2, 'd':3})
s2=Series([-0.9, -1.7, 1.1], index=index[1:])
d = {'A':s1, 'B':s2}
df = DataFrame(d)
df.head()
```

Out[2]:

	A	B
a	0	NaN
b	1	-0.9
c	2	-1.7
d	3	1.1

In [3]:
```
df.reindex(['c','a','b'])
```

Out[3]:

	A	B
c	2	-1.7
a	0	NaN
b	1	-0.9

In [4]:
```
#sort a DF by a (list of) column (s)
df.sort_values('B')
```

Out[4]:

	A	B
c	2	-1.7
b	1	-0.9
d	3	1.1
a	0	NaN

INDEX TO/FROM A COLUMN

In [5]:
```
# Set dataframe column as index
df2 = df.set_index('A')
df2
```

Out[5]:

	B
A	
0	NaN
1	-0.9
2	-1.7
3	1.1

In [6]:
```
# Opposite operation
df2.reset_index()
```

Out[6]:

	A	B
0	0	NaN
1	1	-0.9
2	2	-1.7
3	3	1.1

Dealing with date & time

CREATING DATE/TIME INDEXES

In [1]:
```
from pandas import date_range
from pandas.tseries import offsets
from pandas import Series
from numpy.random import randn
```

In [2]:
```
# The index can be a list of
# dates+times locations that can be
# automatically generated
date_range('1/1/2000',periods=4)
```

Out[2]:
```
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04'], dtype='datetime64[ns]', freq='D')
```

In [3]:
```
# Specify frequency: us,ms,S,T,H,D,B,
# W,M,3min, 2h20min, 2W,...
r=date_range('1/1/2000',periods=72, freq='H')
i=date_range('1/1/2000',periods=4,freq=offsets.YearEnd())
i=date_range('1/1/2000',periods=4,freq='3min')
print(i)
```

```
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 00:03:00',
               '2000-01-01 00:06:00', '2000-01-01 00:09:00'],
              dtype='datetime64[ns]', freq='3T')
```

In [4]:
```
ts=Series(range(4), index=i)
print(ts)
```

```
2000-01-01 00:00:00    0
2000-01-01 00:03:00    1
2000-01-01 00:06:00    2
2000-01-01 00:09:00    3
Freq: 3T, dtype: int64
```

UP-DOWN-SAMPLING

In [5]:
```
ts.resample('T').mean()
```

Out[5]:
```
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    NaN
2000-01-01 00:03:00    1.0
2000-01-01 00:04:00    NaN
2000-01-01 00:05:00    NaN
2000-01-01 00:06:00    2.0
2000-01-01 00:07:00    NaN
2000-01-01 00:08:00    NaN
2000-01-01 00:09:00    3.0
Freq: T, dtype: float64
```

In [6]:
```
# Group hourly data into daily
ts2 = Series(randn(72), index=r)
ts2.resample('D',closed='left', label='left').mean()
```

Out[6]:
```
2000-01-01   -0.238845
2000-01-02    0.273158
2000-01-03    0.009712
Freq: D, dtype: float64
```

Dealing with date & time II

TIME ALIGNMENT

In [1]:
```
from pandas import date_range
from pandas.tseries import offsets
from pandas import DataFrame
from numpy.random import rand
```

In [2]:
```
# Data alignment based on time is one
# of Panda's most celebrated features
daily = date_range('2000-01-01', freq='D', periods=5)
df = DataFrame(rand(5), index=daily, columns=['A'])
df
```

Out[2]:

	A
2000-01-01	0.337957
2000-01-02	0.002200
2000-01-03	0.530821
2000-01-04	0.551186
2000-01-05	0.600895

In [3]:
```
bidaily = date_range('2000-01-01',freq='2D', periods=3)
df2 = DataFrame(rand(3),index=bidaily, columns=['B'])
df2
```

Out[3]:

	B
2000-01-01	0.667605
2000-01-03	0.346022
2000-01-05	0.802351

Concat 2 DataFrames

In [4]:
```
from pandas import concat
concat([df, df2], axis=1)
```

Out[4]:

	A	B
2000-01-01	0.401205	0.943827
2000-01-02	0.265416	NaN
2000-01-03	0.065954	0.374362
2000-01-04	0.201372	NaN
2000-01-05	0.640020	0.564318

TASK02

- Use the Apple stock prices from 2017:

In [1]:
```
import pandas as pd
```

In [2]:
```
aapl = pd.read_table('D:\Tableau\AAPL_2017.csv',sep=',')
```

In [3]:
```
aapl.head()
```

Out[3]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2017-01-03	115.800003	116.330002	114.760002	116.150002	113.847588	28781900
1	2017-01-04	115.849998	116.510002	115.750000	116.019997	113.720161	21118100
2	2017-01-05	115.919998	116.860001	115.809998	116.610001	114.298462	22193600
3	2017-01-06	116.779999	118.160004	116.470001	117.910004	115.572708	31751900
4	2017-01-09	117.949997	119.430000	117.940002	118.989998	116.631294	33561900

1. Print the data from the last 4 weeks (see the .last method)
2. Extract the adjusted close column ("Adj Close"), resample the full data to a monthly period. Do this 3 times, using the min, max, and mean of the resampling window.

Dealing with missing data

Dealing with missing data

- To signal a missing value, Pandas stores a NaN(Not a Number) value defined in Numpy (np.nan)
- Unlike other packages (like Numpy), most operators in Pandas will ignore NaN values in a Pandas data structure.

```
In [1]: import numpy as np
        from pandas import Series

In [2]: a=np.array([1,2,3,np.nan])

In [3]: a.sum()

Out[3]: nan

In [4]: s = Series(a)

In [5]: s.sum()

Out[5]: 6.0
```

Dealing with missing data

FIND MISSING VALUES

In [6]:
```
from pandas import DataFrame, Series
```

In [7]:
```
index=['a', 'b', 'c', 'd']

s1 = Series({'a':1 , 'c':3, 'd':4})

s2=Series([3.5,4.5], index=index[2:])

d = {'A':s1, 'B':s2}
```

In [8]:
```
df = DataFrame(d, index=index)
df
```

Out[8]:

	A	B
a	1.0	NaN
b	NaN	NaN
c	3.0	3.5
d	4.0	4.5

In [9]:
```
#Boolean mask for all null values:
#np.nan and None.
#Use notnull method for the inverse
df.isnull()
```

Out[9]:

	A	B
a	False	True
b	True	True
c	False	False
d	False	False

REMOVE/REPLACE NaN

In [10]:
```
#Replace missing values manually
from pandas import isnull
df[isnull(df)]=0
```

In [11]:
```
df
```

Out[11]:

	A	B
a	1.0	0.0
b	0.0	0.0
c	3.0	3.5
d	4.0	4.5

In [12]:
```
#Inverse operation
df[df == 0] = np.nan
df
```

Out[12]:

	A	B
a	1.0	NaN
b	NaN	NaN
c	3.0	3.5
d	4.0	4.5

In [13]:
```
#Fill na from previous value
df.fillna(method='ffill')
```

Out[13]:

	A	B
a	1.0	NaN
b	1.0	NaN
c	3.0	3.5
d	4.0	4.5

In [14]:
```
#Remove all rows with missing values
df.dropna(how='all')
```

Out[14]:

	A	B
a	1.0	NaN
c	3.0	3.5
d	4.0	4.5

In [15]:
```
df.dropna(how='any')
```

Out[15]:

	A	B
c	3.0	3.5
d	4.0	4.5

Dealing with missing data

interpolation is a method of constructing new data points within the range of a discrete set of known data points.

Interpolate Nans away

In [16]:
```
df.interpolate()
```

Out[16]:

	A	B
a	1.0	NaN
b	2.0	NaN
c	3.0	3.5
d	4.0	4.5

In [ ]:
```
from pandas import DataFrame
import numpy as np
```

In [ ]:
```
df = DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
 'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})
```

In [ ]:
```
df.interpolate()
```

In [ ]:
```
df.interpolate(method='barycentric')
```

In [ ]:
```
df.interpolate(method='pchip')
```

In [ ]:
```
df.interpolate(method='akima')
```

In [ ]:
```
df.interpolate(method='polynomial', order=2)
```

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

Computations and statistics

Computations and statistics

- **Rule 1:** Mathematical operators (+ - * / exp, log, …) apply element by element, on the values.
- **Rule 2:** Reduction operations (mean, std, skew, kurt, sum, prod, …) are applied column by column
- **Rule 3:** Operations between multiple Pandas object implement auto-alignment based on index first

Computations with Pandas

Computations are applied

Column-by-column

In [1]:
```
from pandas import DataFrame, Series
```

In [2]:
```
index=['a', 'b', 'c', 'd']

s1 = Series({'a':1 ,'b':2, 'c':3, 'd':4})

s2 = Series([-0.9,-1.7,1.1], index=index[1:])

s3 = Series([False, False, False, True], index=index)

d = {'A':s1, 'B':s2 , 'Flags':s3}
```

In [3]:
```
df = DataFrame(d, index=index)
df
```

Out[3]:

	A	B	Flags
a	1	NaN	False
b	2	-0.9	False
c	3	-1.7	False
d	4	1.1	True

In [4]:
```
df.sum()
```

Out[4]:
```
A        10.0
B        -1.5
Flags     1.0
dtype: float64
```


In [4]:
```
df.sum()
```

Out[4]:
```
A        10.0
B        -1.5
Flags     1.0
dtype: float64
```


In [5]:
```
df-1
```

Out[5]:

	A	B	Flags
a	0	NaN	-1
b	1	-1.9	-1
c	2	-2.7	-1
d	3	0.1	0

In [6]:
```
np.exp(df['A'])
```

Out[6]:
```
a     2.718282
b     7.389056
c    20.085537
d    54.598150
Name: A, dtype: float64
```

Statistical Analysis

DESCRIPTIVE STATS

In [8]:
```
df
```
Out[8]:

	A	B	Flags
a	1	NaN	False
b	2	-0.9	False
c	3	-1.7	False
d	4	1.1	True

In [9]:
```
df.mean()
```
Out[9]:
```
A        2.50
B       -0.50
Flags    0.25
dtype: float64
```

In [11]:
```
df.mean(axis=1)
```
Out[11]:
```
a    0.500000
b    0.366667
c    0.433333
d    2.033333
dtype: float64
```

In [13]:
```
#min/max location (series only)
df['B'].idxmin()
```
Out[13]: 'c'

In [14]:
```
df.describe()
```
Out[14]:

	A	B
count	4.000000	3.000000
mean	2.500000	-0.500000
std	1.290994	1.442221
min	1.000000	-1.700000
25%	1.750000	-1.300000
50%	2.500000	-0.900000
75%	3.250000	0.100000
max	4.000000	1.100000

Data Filtering and Aggregation

Split, apply and combine

- It is often necessary to apply different operations on different subgroups
  - Traditionally handled by SQL-based systems
  - Pandas provides in-memory, sql-like set of operations
- General ‘framework’: split, apply, combine :
  - Splitting the data into groups (based on some criterion, e.g. column value)
  - Applying a function to each group independently
  - Combine the results back into a data structure (e.g. dataframe)

Data aggregation: Split

In [1]:
```
from pandas import DataFrame, Series
import numpy as np
```

In [2]:
```
index=['a', 'b', 'c', 'd']

s1 = Series({'a':1 ,'b':2, 'c':3, 'd':4})

s2 = Series([-0.9,-1.7,1.1], index=index[1:])

s3 = Series([False, False, False, True], index=index)

d = {'A':s1, 'B':s2 , 'Flags':s3}
```

In [3]:
```
df = DataFrame(d, index=index)
df
```

Out[3]:

	A	B	Flags
a	1	NaN	False
b	2	-0.9	False
c	3	-1.7	False
d	4	1.1	True

Group data by one column's value

In [4]:
```
gb = df.groupby('Flags')
```

gb is a groupby object

In [5]:
```
gb.groups
```

Out[5]:
```
{False: Index(['a', 'b', 'c'], dtype='object'),
 True: Index(['d'], dtype='object')}
```

In [6]:
```
# gb = iterator of tuples with
# group name and sub part of df
```

In [7]:
```
for value, subdf in gb:
    print (value)
    print (subdf)
```

```
False
   A    B  Flags
a  1  NaN  False
b  2 -0.9  False
c  3 -1.7  False
True
   A    B  Flags
d  4  1.1   True
```

Data aggregation: Apply

APPLY WITH aggregate() or agg()

In [8]:
```
gb.sum()
```

Out[8]:

	A	B
Flags		
False	6	-2.6
True	4	1.1

In [10]:
```
# More flexible but slower
summed = gb.aggregate(np.sum)
summed
```

Out[10]:

	A	B
Flags		
False	6	-2.6
True	4	1.1

In [11]:
```
# Given a list or dict
gb.agg([np.mean, np.std])
```

Out[11]:

	A		B	
	mean	**std**	**mean**	**std**
Flags				
False	2	1.0	-1.3	0.565685
True	4	NaN	1.1	NaN

In [12]:
```
gb.agg({'A':'sum', 'B':'std'})
```

Out[12]:

	B	A
Flags		
False	0.565685	6
True	NaN	4

Combining tables

Merging

pandas.merge connects **DataFrames** based on one or more keys (close to SQL join).

Let's assume we are running a restaurant, and store customer information and orders coming in in different tables:

In [1]:
```
from pandas import DataFrame , merge
```

In [2]:
```
customers = DataFrame({'id': range(3), 'name': ['john', 'alex', 'lucy']})
orders = DataFrame({'id': [1, 0, 1, 2], 'order': ['pasta', 'salad', 'coke', 'fries']})
```

In [3]:
```
print(customers)
```

```
   id  name
0   0  john
1   1  alex
2   2  lucy
```

In [4]:
```
print(orders)
```

```
   id  order
0   1  pasta
1   0  salad
2   1   coke
3   2  fries
```

In [5]:
```
merge(customers, orders, on='id')
```

Out[5]:

	id	name	order
0	0	john	salad
1	1	alex	pasta
2	1	alex	coke
3	2	lucy	fries

Merging (Cont.)

- OUTER vs INNER JOINS

In [6]:
```
orders = orders.append({'id': 3, 'order': 'pasta'}, ignore_index=True)
print(orders)
```

```
   id  order
0   1  pasta
1   0  salad
2   1   coke
3   2  fries
4   3  pasta
```

In [7]:
```
merge(customers, orders, on='id')
```

Out[7]:

	id	name	order
0	0	john	salad
1	1	alex	pasta
2	1	alex	coke
3	2	lucy	fries

In [8]:
```
merge(customers, orders, on='id', how='outer')
```

Out[8]:

	id	name	order
0	0	john	salad
1	1	alex	pasta
2	1	alex	coke
3	2	lucy	fries
4	3	NaN	pasta

Data summarization

Pivot tables

- A **pivot table** is a table that summarizes data in another table, and is made by applying an operation such as **sorting**, **averaging**, or **summing** to data in the first table, typically including **grouping** of the data.

In [1]:
```python
from pandas import Series, DataFrame ,to_datetime
```

In [2]:
```python
index = range(6)
date = Series([ '2000-01-03','2000-01-04','2000-01-05','2000-01-03','2000-01-04','2000-01-05'], index=index)
variable = Series(['A','A','A','B','B','B'],index=index)
value = Series([0.469112,-0.282863, -1.509059, -1.135632, 1.212112, -0.173215] , index=index)
```

In [3]:
```python
# convert to date
date = to_datetime(date)
```

In [4]:
```python
d = {'date':date, 'variable':variable, 'value':value}
df = DataFrame(d)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 3 columns):
date        6 non-null datetime64[ns]
value       6 non-null float64
variable    6 non-null object
dtypes: datetime64[ns](1), float64(1), object(1)
memory usage: 224.0+ bytes
```

In [5]:
```python
df
```

Out[5]:

	date	value	variable
0	2000-01-03	0.469112	A
1	2000-01-04	-0.282863	A
2	2000-01-05	-1.509059	A
3	2000-01-03	-1.135632	B
4	2000-01-04	1.212112	B
5	2000-01-05	-0.173215	B

In [6]:
```python
df.pivot(index='date',columns='variable', values='value')
```

Out[6]:

variable	**A**	**B**
date		
2000-01-03	0.469112	-1.135632
2000-01-04	-0.282863	1.212112
2000-01-05	-1.509059	-0.173215

In [7]:
```python
import pandas as pd
import numpy as np
```

In [8]:
```python
data = pd.read_csv('pivot.csv')
```

In [9]:
```python
print(data)
```

```
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
2  foo  one  large  2
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

In [10]:
```python
table= data.pivot_table(index=['A', 'B'], columns=['C'], values='D', aggfunc=np.sum)
```

In [11]:
```python
table
```

Out[11]:

	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

reference

- https://github.com/jonathanrocher/pandas_tutorial
- https://github.com/chendaniely/scipy-2017-tutorial-pandas
- https://pandas.pydata.org/pandas-docs/stable/index.html