ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Analyzing and Manipulating data with Pandas

Enthought, Inc.
www.enthought.com

Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin, TX 78701

www.enthought.com

Q2-2015
letter
04e5964

Analyzing and Manipulating data with Pandas

Enthought, Inc.
www.enthought.com

Data Analysis with pandas

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

Pandas

Pandas is a library that makes the analysis of complex, tabular datasets ***easy***.

FEATURES

- Defines **tabular data types**: database-like tables, with labelled rows and columns;
- **Data consolidation and data integration**: remove duplicates, clean data, manage missing values; automatically align tables by index;
- **Summarization**: create “pivot” tables;
- **In-memory, SQL-like operations**: join, aggregate (group by);
- Very flexible **import/export** of data;
- **Date and time** handling built-in, including timezones;
- **Easy visualization** based on Matplotlib.

ENTHOUGHT

Pandas data I/O

ENTHOUGHT

Pandas data I/O

Pandas provides a high-level interface to and from many file formats used in data science:
`.txt`, CSV, json, HTML, clipboard, Excel (`.xls .xlsx`), pickle, HDF5, SQL, R (exp.), Stata `.dta`, ...

For any given format, there is

- a `read_**` function,
- a `to_**` method attached to all Pandas data objects.

You might need to install other libraries for some of the formats (Pandas will warn you if that is the case).

ENTHOUGHT

Pandas' `read_table` example

FEATURES

`read_table` can read tabular text (for example CSV files) into a `DataFrame` and implements the following:

- detect comments, headers and footers
- specify which column is the index
- specify the column names 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
- ...

EXAMPLE

```
# Historical_data.csv
Date,AAPL,GOOGL,MSFT,PG,XOM
2005-01-03,64.78,197.4,26.8,-,51.02
2005-01-04,63.79,201.4,26.87,55.12,50.34
2005-01-05,64.46,193.45,26.84,55.28,49.83
...
```

```
>>> read_table('historical_data.csv',
               sep=',', header=1,
               index_col=0,
               na_values=['-']
               parse_dates=True)
              AAPL    GOOGL   MSFT     PG     XOM
Date
2005-01-03   64.78   197.40   26.80    NaN   51.02
2005-01-04   63.79   201.40   26.87  55.12   50.34
2005-01-05   64.46   193.45   26.84  55.28   49.83
...
```


ENTHOUGHT

Reading large files in chunks

Pandas supports reading potentially very large files in chunks, e.g.:

```
>>> chunks = []
>>> reader = pd.read_csv('contributions_2012.csv',
...                      chunksize=100000)
>>> for table in reader:
...     new_yorkers = table['contbr_city'] == 'NEW YORK'
...     chunks.append(table[new_yorkers])
>>> new_york_contributions = pd.concat(chunks)
>>> print len(new_york_contributions)
25858

>>> print new_york_contributions.iloc[143]
cmte_id                      C00431171
cand_id                      P80003353
cand_nm                   Romney, Mitt
contbr_st                           NY
contbr_occupation            EXECUTIVE
contb_receipt_amt                 2500
contb_receipt_dt             22-JUN-11
...
```

ENTHOUGHT

Pandas IO summary

READING

Format	Method, Function, Class
txt, csv	read_table, read_csv
pickle	read_pickle
HDF5	read_hdf, HDFStore
SQL	read_sql_table
Excel	read_excel
R (exp.)	rpy.common.load_data

WRITING

Format	Method, Function, Class
txt, csv	to_string, to_csv
html	to_html
pickle	to_pickle
HDF5	to_hdf, HDFStore
Excel	to_excel, ExcelWriter
R (exp.)	rpy.common.convert_to_r_dataframe

EXAMPLES

```
# Excel
>>> writer = ExcelWriter('out.xlsx')
>>> df1.to_excel(writer, 'Sheet1')
>>> df2.to_excel(writer, 'Sheet2')
>>> writer.save()
>>> read_excel("out.xlsx", "Sheet2")

# HDF5
>>> stor = HDFStore('foo.h5')
>>> stor['ser1'] = s
>>> s2 = stor['ser1']
>>> stor.close()
>>> s3 = read_hdf('foo.h5', 'ser1')

# Scrape tables from HTML webpages
>>>
read_html("http://www.bloomberg.com/mar
kets/world/")
```


ENTHOUGHT

Storing data in Pandas

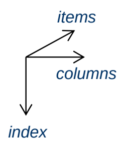
ENTHOUGHT

New Data Structures

PANDA = PANel DAta = 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`.



TAMU

UT

	Rk,	Fst N,	Lst N,	DoB	Gend	Grade	offset
1	4	John	Doe	1981	M	4.18	-1.18
2	2	Jill	Ford	1975	F	5.26	1.26
3	5	Chris	Jones	NaN	M	3.91	0.91
4	6	Dave	Smith	1965	M	1.23	NaN
5	1	Ellen	Frank	1973	F	5.52	0.52
6	3	Frank	Hart	1976	M	4.71	-0.71

ENTHOUGHT

Series

Definition

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

```
>>> from pandas import *
>>> s = Series({'a':0,'b':1,'c':2,'d':3})
# Dict-like access can be label-based
>>> s['b']
1
```

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

ENTHOUGHT

Creating Series

FROM LIST AND DICT

```
# Data and corresponding indices can
# be stored in lists.
>>> index = ['a', 'b', 'c', 'd']
>>> Series(range(4), index=index,
           name='first series')
a     0
b     1
c     2
d     3
Name: first series
# data + indices in a dict
>>> d = {'a':0,'b':1,'c':2,'d':3}
>>> s = Series(d, name='first series')
>>> s.index
Index([a, b, c, d], dtype=object)
>>> s.values, type(s.values)
array([0, 1, 2, 3], dtype=int64)
numpy.ndarray
>>> s.dtype
dtype('int64')
```

FROM A NUMPY ARRAY

```
>>> from numpy.random import randn
>>> Series(randn(4), index=index)
a   -1.062984
b   -0.961625
c   -0.720323
d    0.336753
```

ACCESS OR ADD ELEMENTS

```
# Request existing values
>>> s['b']
1
# Modify an existing value
>>> s['b'] = 3
# Add new elements
>>> s['e'] = 5
>>> s
a    0
b    3
c    2
d    3
e    5
```


ENTHOUGHT

Dataframes

DEFINITION

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’

```
>>> s1 = Series({'a': 1, 'b': 2, 'c': 3})
>>> s2 = Series({'a': True, 'b': False, 'c': True})
>>> df = DataFrame({'col1': s1, 'col2': s2})
# Dict-like access is column-based
>>> df['col1']
a     1
b     2
c     3
Name: col1, dtype: int64
```

Creating DataFrames

FROM A DICT OF SERIES

```
# DF from a dict of series: keys are
# column names.
>>> s2=Series([-0.9, -1.7, 1.1],
              index=index[1:])
>>> d = {'A':s, 'B':s2}
>>> df = DataFrame(d)
   A     B
a  0   NaN
b  1  -0.9
c  2  -1.7
d  3   1.1
>>> df.index, df.columns
Index([a, b, c, d], dtype=object)
Index([A, B], dtype=object)
>>> df.shape, df.dtypes
(4,2)
A          int64
B        float64
>>> df.values
array([[ 0.   ,  nan],
       [ 1.   , -0.9],
       [ 2.   , -1.7],
       [ 3.   ,  1.1]])
```

FROM A NUMPY ARRAY

```
>>> DataFrame(randn(4,4), index=index,
    columns=['A','B','C','D'])
         A        B        C        D
a  0.28164 -0.36826  0.04011  1.25030
b -0.71049 -1.23956 -0.08504 -0.08336
c -1.29446  0.70709  1.39642  0.49035
d  0.74632 -0.03512 -0.69237  0.81488
```

ACCESS OR ADD COLUMNS

```
# Columns accessed like a dict...
>>> col1 = df['A']
# Create a new column
>>> df['Flag'] = df['B'] > 0
>>> df
   A    B    Flag
a  0   NaN   False
b  1  -0.9   False
c  2  -1.7   False
d  3   1.1    True
```

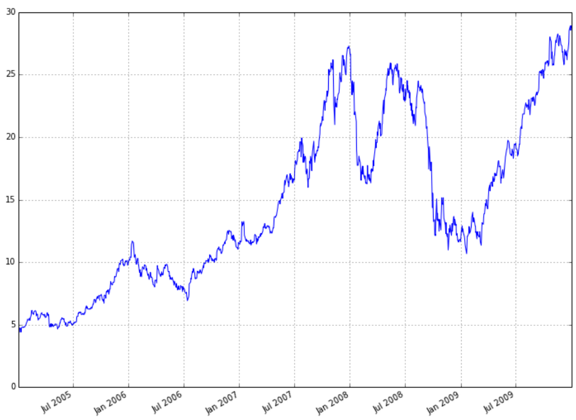

Visualizing Series and DataFrames

ENTHOUGHT

Visualizing (Time)`Series`

```
>>> ts = df['AAPL']
>>> ts.plot()
```

```
>>> ts2 = df['MSFT']
>>> ts2.plot(secondary_y=True)
```



To follow along these slides, pull out the `pandas_plotting/` demo.

ENTHOUGHT

Gaining more control

Control the output of the plot methods with keyword arguments, including:

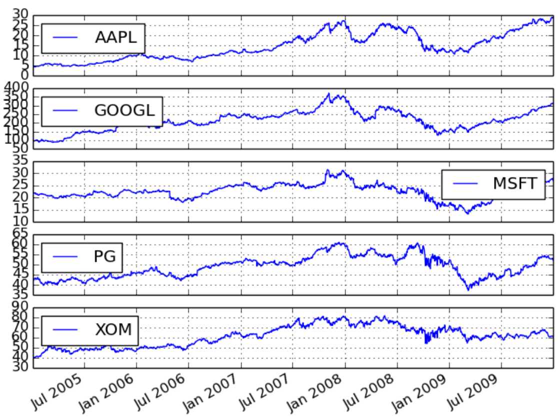
- `kind : {'line', 'bar', 'barh', 'kde', 'density', 'area'}`
- `logx, logy, loglog : {True, False}`
- `xlim, ylim`
- `style (for e.g. 'g-*' for a green line with stars at points)`
- `figsize, label, ...`

Greater control is possible with Matplotlib:

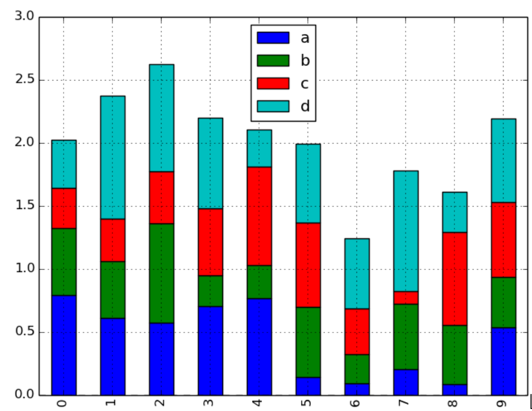
```
>>> import matplotlib.pyplot as plt
>>> plt.figure() # Create a new figure
>>> plt.title('Best plot ever')
>>> plt.ylabel('Adj Close price')
>>> plt.legend()
>>> plt.savefig() # Save to png, jpeg, eps, svg, ...
```

Visualizing `DataFrame`s

```
# By default, DF plot method
# draws a line for each col
>>> df.plot(subplots=True)
```

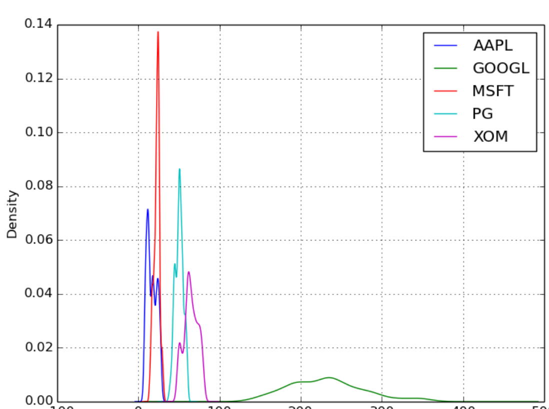



```
# Use stacked bar plots to show
# proportions varying in time.
>>> df2 = DataFrame(rand(10,4),
        columns=list('abcd'))
>>> df2.plot(kind='bar',
             stacked=True)
```

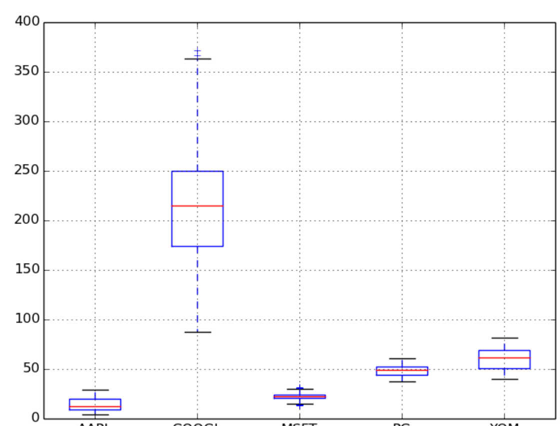


Visualizing Distributions

```
# The kde mode will build an
# estimation of the pdf for
# each series.
>>> df.plot(kind="kde")
```



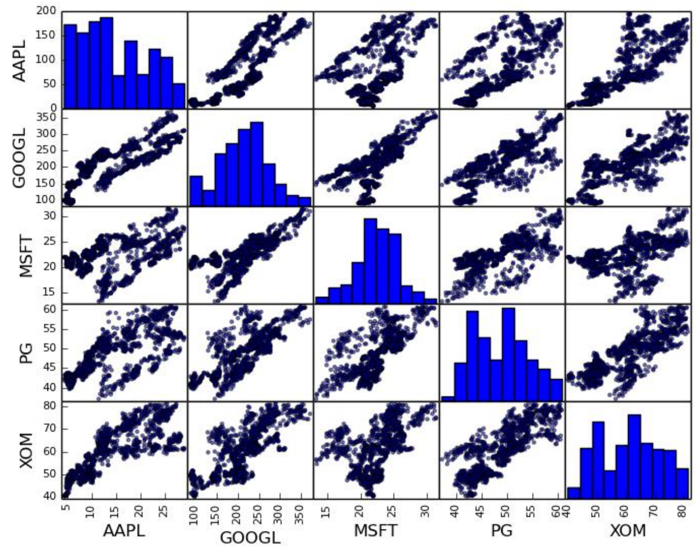

```
# Similar info with a boxplot
>>> df.boxplot()
```



ENTHOUGHT

Visualizing Correlations

```
# Search for correlations between all columns of a DF
>>> from pandas.tools.plotting import scatter_matrix
>>> scatter_matrix(df)
```



ENTHOUGHT

More visualization?

See also:

>>> pandas.tools.plotting.<TAB>

and in particular **`lag_plot`**, **`autocorrelation_plot`**, **`andrews_curves`**.

http://pandas.pydata.org/pandas-docs/dev/visualization.html

http://matplotlib.org/

http://matplotlib.org/gallery.html

Accessing values in Pandas: Indexing

Pandas and indexing

`Series` and `DataFrame`s 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 (more on this later)

Essentially:

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

```
>>> s = Series({'a': 0, 'b': 1,
                'c': 2})
>>> s['a']
0
>>> df = DataFrame({'A': s, 'B': -s})
>>> df['A']
a 0
b 1
c 2
```

```
# BUT if you do slicing
>>> df[:2] # first two rows !!
   A  B
a  0  0
b  1 -1
```

Pandas and indexing (Cont.)

LABEL-BASED VS POSITION-BASED INDEXING

Indexing operator `[]` has an ambiguity:

- `Series[integer_value]`: position or label?
- `DataFrame[integer_value]`: position or column name ?

API is a bit messy here, greatly improved in versions >= 0.11.0:

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

```
>>> s = Series({'a': 0, 'b': 1, 'c': 2})
>>> s.iloc[1]
1
>>> s.iloc['a']
TypeError: the label [a] is not a proper indexer for this index ...
>>> s.loc['a']
0
>>> s.loc[0]
KeyError: 'the label [0] is not in the [index]'
```


Indexing into `Series`

ACCESSING 1 ELEMENT

```
>>> index = ['a', 'b', 'c', 'd']
>>> s = Series(range(4), index=index)
# Access elements based on position
>>> s.iloc[2]
2
# Access elements based on label
>>> s.loc['c']
2
# Indexing into a Series is equivalent
>>> s['c']
2
```

SLICING ELEMENTS OUT

Form: `s.iloc[pos_lower:pos_upper:step]`

```
>>> s.iloc[:2]
a     0
b     1
# Every other element
>>> s.iloc[::2]
a     0
c     2
```

Form: `s.loc[label_lower:label_upper:step]`

```
>>> s.loc['a':'c']
a     0
b     1
c     2 # upper limit included!
```

FANCY-INDEXING

```
# Custom selection of elements
>>> s[[True, False, True, True]]
a     0
c     2
d     3

# Masks can be created by comparing
# values in the Series or another one
>>> s>1
a     False
b     False
c      True
d      True
>>> s[s>1]
c     2
d     3
```

ENTHOUGHT

Indexing into DataFrames

ACCESS ELEMENTS

```
>>> df
   A     B
a  0   NaN
b  1  -0.9
c  2  -1.7
d  3   1.1
# 1 (or more) column accessed like a
# dict...
>>> df['A']
a  0
b  1
c  2
d  3
... or like an object
>>> series2 = df.B
# Access all columns for 1 index
>>> df.loc['c']
A            2
B         -1.7
Name: c, dtype: float64
# or 1 element of the table
>>> df.loc['c','B']
-1.7
```

SLICING ELEMENTS OUT

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

```
>>> sub_df = df.loc["c":, "A":"B"]

# Incomplete slicing assumes all
# elements in other dimensions.
>>> df.loc["c":]
   A    B
c  2 -1.7
d  3  1.1
```

MIXED INDEXING

Mixed indexing using `.ix`:

```
>>> sub_df = df.ix[2, "B"]
-1.7
```


ENTHOUGHT

Give it a try!

Create a DataFrame with random data:

```
import pandas as pd
import numpy as np
data = np.arange(12).reshape(4, 3)
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’
5. Plot the data as a box plot (`kind='box'`)

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`

RE-INDEXING

```
>>> index = ['a', 'b', 'c', 'd']
>>> s = Series(range(4), index=index)

# Select a different set of indices
>>> s.reindex(['c', 'b', 'a', 'e'])
c     2
b     1
a     0
e   NaN

# Sort by values. See s.sort_index()
# to sort based on index value.
>>> s.order(ascending=False)
s    3
r    2
q    1
p    0
```

ALIGNMENT OF 2 SERIES

```
>>> s = Series(range(4), index=index)
>>> s2 = s.iloc[:-2]
>>> s2
a    0
b    1

# Operations automatically align on
# the index (different from NumPy)
>>> s + s2
a     0
b     2
c   NaN
d   NaN
```

Re-indexing DataFrames

RE-INDEXING DATAFRAMES

```
>>> df
   A    B  flags
a  0  NaN  False
b  1 -0.9  False
c  2 -1.7  False
d  3  1.1   True
>>> df.reindex(['c', 'a', 'b'])
   A    B  flags
c  2 -1.7  False
a  0  NaN  False
b  1 -0.9  False
# Sort a DF by a (list of) column(s)
>>> df.sort('B')
   A    B  flags
c  2 -1.7  False
b  1 -0.9  False
d  3  1.1   True
a  0  NaN  False
```

INDEX TO/FROM A COLUMN

```
# Set dataframe column as index
>>> df2 = df.set_index('A')
>>> df2
     B  flags
A
0  NaN  False
1 -0.9  False
2 -1.7  False
3  1.1   True

# Opposite operation
>>> df2.reset_index()
   A    B  flags
0  0  NaN  False
1  1 -0.9  False
2  2 -1.7  False
3  3  1.1   True
```


Dealing with date & time

CREATING DATE/TIME INDEXES

```
# The index can be a list of
# dates+times locations that can be
# automatically generated
>>> date_range('1/1/2000',periods=4)
<class
'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-04
00:00:00]
Length: 4, Freq: D, Timezone: None
# 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=datetools.YearEnd())
>>> i=date_range('1/1/2000',periods=4,
...       freq='3min')
>>> ts=Series(range(4), index=i)
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
```

UP-/DOWN-SAMPLING

```
>>> ts.resample('T')
2000-01-01 00:00:00      0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    NaN
2000-01-01 00:03:00      1
2000-01-01 00:04:00    NaN
2000-01-01 00:05:00    NaN
2000-01-01 00:06:00      2
2000-01-01 00:07:00    NaN
2000-01-01 00:08:00    NaN
2000-01-01 00:09:00      3
Freq: T

# Group hourly data into daily
>>> ts2 = Series(randn(72), index=r)
>>> ts2.resample('D', how='mean',
...         closed='left', label='left')
01-Jan-2000    0.397501
02-Jan-2000    0.186568
03-Jan-2000    0.327240
Freq: D
```

ENTHOUGHT

Dealing with date & time II

TIME ALIGNMENT

```
# 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(random.rand(5),
...     index=daily, columns=['A'])
>>> df
                   A
2000-01-01  0.954140
2000-01-02  0.511243
2000-01-03  0.979188
2000-01-04  0.793727
2000-01-05  0.238190
>>> bidaily = pd.date_range('2000-01-01',
...      freq='2D', periods=3)
>>> df2 = pd.DataFrame(np.random.rand(3),
...      index=bidaily, columns=['B'])
>>>df2
                   B
2000-01-01  0.007215
2000-01-03  0.797108
2000-01-05  0.440173
```

TIME ALIGNMENT (cont.)

```
>>> concat([df, df2], axis=1)
                   A         B
2000-01-01  0.954140  0.007215
2000-01-02  0.511243       NaN
2000-01-03  0.979188  0.797108
2000-01-04  0.793727       NaN
2000-01-05  0.238190  0.440173
```


ENTHOUGHT

Give it a try!

Download the Apple stock prices from 2010:

```
import pandas.io.data as web
aapl =
web.get_data_yahoo('AAPL','1/1/2010')
```

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 and plot. Do this 3 times, using the min, max, and mean of the resampling window.

Dealing with missing data

Dealing with missing data

PANDAS PHILOSOPHY

- 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 datastructure.

```
>>> import numpy as np
>>> a = np.array([1,2,3,np.nan])
>>> a.sum()
nan
>>> s = Series(a)
>>> s.sum()
6
```

ENTHOUGHT

Dealing with missing data

FIND MISSING VALUES

```
>>> df
      s1   s2
a      1  NaN
b    NaN  NaN
c      3  3.5
d      4  4.5
# Boolean mask for all null values:
# np.nan and None .
# Use notnull method for the inverse
>>> df.isnull()
       s1     s2
a   False   True
b    True   True
c   False  False
d   False  False
```

REMOVE/REPLACE NaN

```
# Replace missing values manually
>>> df[isnull(df)] = 0.
```

```
# Inverse operation
>>> df[df == 0] = np.nan
# Fill na from previous value
>>> df.fillna(method='ffill')
   s1   s2
a   1  NaN
b   1  NaN
c   3  3.5
d   4  4.5
# Remove all rows w/ missing values
>>> df.dropna(how='all')
   s1   s2
a   1  NaN
c   3  3.5
d   4  4.5
>>> df.dropna(how='any')
   s1   s2
c   3  3.5
d   4  4.5
# Interpolate NaNs away
>>> df.interpolate()
   s1   s2
a   1  NaN
b   2  NaN
c   3  3.5
d   4  4.5
```


ENTHOUGHT

Give it a try!

Download the Apple stock prices from 2010:

```
import pandas.io.data as web
aapl =
web.get_data_yahoo('AAPL','1/1/2010')
```

1. Extract the adjusted close column and plot it.
2. Resample to a 6 hours period, then interpolate through the missing values using cubic interpolation. Plot the resulting time series.

ENTHOUGHT

Computations and statistics

ENTHOUGHT

Computations with DataFrames

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.

ENTHOUGHT

Computations with Pandas

```
# Computations are applied
# column-by-column
>>> df
   A      B  Flags
a  0    NaN  False
b  1   -0.9  False
c  2   -1.7  False
d  3    1.1   True

>>> df.sum()
A          6.0
B         -1.5
Flags      1.0
dtype: float64

# Adding a series or re-scaling
>>> row = df.iloc[1]
>>> df - row
    A      B   Flag
a  -1    NaN  False
b   0      0  False
c   1   -0.8  False
d   2    2.0   True
```

DATAFRAME REDUCTION

```
# 'apply' a custom function to
# columns. The function receives a
# column (Series) and returns a value
>>> f = lambda x: x.max() - x.min()
>>> df.apply(f, axis=0)
A      3.0
B      2.8
Flags 1.0
```

DATAFRAME TRANSFORMATION

```
# applymap is similar but receives a
# value and return a value.
>>> df.applymap(lambda x: len(str(x)))
   A  B  Flags
a  1  3      5
b  1  4      5
c  1  4      5
d  1  3      4
```


ENTHOUGHT

Statistical Analysis

DESCRIPTIVE STATS

```
>>> df
   A       B    Flag
a  0     NaN   False
b  1    -0.9   False
c  2    -1.7   False
d  3     1.1    True
# Descriptive stats available:
# count, sum, mean, median, min, max,
# abs, prod, std, var, skew, kurt,
# quantile, cumsum, cumprod, cummax
# Stats on DF are column per column
>>> df.mean()
A         1.50
B        -0.50
flag      0.25
>>> df.mean(axis=1)
a     0.000000
b     0.033333
c     0.100000
d     1.700000
# min/max location (Series only)
>>> df['B'].argmin()
'c'
```

```
>>> df.describe()
             A          B    Flag
count  4.000000   3.000000      4
mean   1.500000  -0.500000   0.25
std    1.290994   1.442221    0.5
min    0.000000  -1.700000  False
25%    0.750000  -1.300000      0
50%    1.500000  -0.900000      0
75%    2.250000   0.100000   0.25
max    3.000000   1.100000   True
```

WINDOWED STATS

```
# Available rolling stats discoverable
# in IPython using
>>> pd.rolling_<TAB>
# Includes std, min, max, count, sum
# quantile, kurt, skew, ...
# For example,
>>> t = pd.rolling_mean(s, window=20)
# Custom function on ndarray possible
>>> f = lambda x: return x.mean()
>>> t == pd.rolling_apply(s, 20, f)
True
```

ENTHOUGHT

Correlations

CORRELATIONS

```
# Correlation of Series
>>> ts.corr(ts2)
0.066666666666666693

# Pair-wise correlations of the columns.
# Optional argument: 'method', one of
# {'pearson', 'kendall', 'spearman'}
>>> corr_matrix = df.corr()

# Pair-wise covariance of the columns
>>> cov_matrix = df.cov()
```

For more stats, see `statsmodels`.

ENTHOUGHT

Data Filtering and Aggregation

Split, apply and combine

RATIONALE

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 (Hadley Wickham, R programmer):
- 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

SPLIT WITH groupby()

```
>>> df
   A    B   Flag
a  0  NaN  False
b  1 -0.9  False
c  2 -1.7  False
d  3  1.1   True
e  4  0.5   True
# Group data by one column’s value
>>> gb = df.groupby('Flag')
# gb is a groupby object
>>> gb.groups
{False: ['a', 'b', 'c'], True: ['d', 'e']}
# gb = iterator of tuples with
# group name and sub part of df
>>> for value, subdf in gb:
        print value
        print subdf
```

```
False
   A    B   Flag
a  0  NaN  False
b  1 -0.9  False
c  2 -1.7  False
True
   A    B  Flag
d  3  1.1  True
e  4  0.5  True
# Displays a subplot per group.
>>> gb.boxplot(column=["A", "B"])
```

groupby() ON THE INDEX

```
>>> df2 = df.reset_index()
>>> even = lambda x: x%2 == 0
>>> gb2 = df2.groupby(even)
>>> gb2.groups
{False: [1, 3], True: [0, 2, 4]}
```

ENTHOUGHT

Data aggregation: Apply

Three ways to apply: `aggregate` (or equivalently `agg`) if each series in each group is turned into one value, `transform` if each series in each group is modified but retains its length, or `apply` in the most general case.

APPLY WITH aggregate() or agg()

```
>>> gb.sum()
       A    B
Flag
False  3 -2.6
True   7  1.6
# More flexible but slower
>>> summed = gb.aggregate(np.sum)
# Given a list or dict
>>> gb.agg([np.mean, np.std])
          A                B
       mean       std   mean       std
Flag
False   1.0  1.000000   -1.3  0.565685
True    3.5  0.707107    0.8  0.424264
```

```
>>> gb.agg({'A':'sum', 'B':'std'})
       A         B
Flag
False  3  0.565685
True   7  0.424264
```

APPLY WITH transform()

```
>>> f = lambda x: x - x.mean()
>>> gb.transform(f)
     A    B
a -1.0  NaN
b  0.0  0.4
c  1.0 -0.4
d -0.5  0.3
e  0.5 -0.3
```


ENTHOUGHT

Data aggregation II

APPLY WITH apply()

```
# Computations from values in groups can be turned into a DF of calcs
>>> desc = lambda x: x.describe()
>>> gb['A'].apply(desc).unstack()
       count  mean       std  min   25%  50%   75%  max
Flag
False      3   1.0  1.000000    0  0.50  1.0  1.50    2
True       2   3.5  0.707107    3  3.25  3.5  3.75    4
>>> f = lambda group: DataFrame({'original':group,
        'demeaned': group - group.mean()})
>>> gb['A'].apply(f)
   demeaned  original
a      -1.0         0
b       0.0         1
c       1.0         2
d      -0.5         3
e       0.5         4
```

ENTHOUGHT

Combining tables

ENTHOUGHT

Merging

Definition

`pandas.merge` connects `DataFrame`s 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:

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

Let's now assume we want to connect customer names to their order. We need to use their id to make that connection:

```
>>> merge(customers, orders, on='id')
   id  name  order
0   0  john  salad
1   1  alex  pasta
2   1  alex   coke
3   2  lucy  fries
```

ENTHOUGHT

Merging (Cont.)

OUTER vs INNER JOINS

```
# Assume a mysterious order comes in
>>> orders = orders.append({'id': 3, 'order': 'pasta'}, ignore_index=True)
```

```
>>> merge(customers, orders, on='id')
   id  name  order
0   0  john  salad
1   1  alex  pasta
2   1  alex   coke
3   2  lucy  fries
```

```
>>> merge(customers, orders, on='id',
          how='outer')
   id  name  order
0   0  john  salad
1   1  alex  pasta
2   1  alex   coke
3   2  lucy  fries
4   3   NaN  pasta
```

Merge method	SQL Join Name	Description
`inner (default)`	INNER JOIN	Use intersection of keys from both frames
`outer`	FULL OUTER JOIN	Use union of keys from both frames
`left`	LEFT OUTER JOIN	Use keys from left frame only
`right`	RIGHT OUTER JOIN	Use keys from right frame only

ENTHOUGHT

Data summarization

ENTHOUGHT

Pivot tables

PIVOTING

```
# Repeating columns can be viewed as
# an additional axis
>>> df
         date variable     value
0  2000-01-03        A  0.469112
1  2000-01-04        A -0.282863
2  2000-01-05        A -1.509059
3  2000-01-03        B -1.135632
4  2000-01-04        B  1.212112
5  2000-01-05        B -0.173215
>>> df.pivot(index='date',
  columns='variable', values='value')
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
```


ENTHOUGHT

Pivot tables II

PIVOT_TABLE

```
# Another way to reshape a DF and
# aggregate at the same time
>>> df
   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

>>> table= df.pivot_table(
   index=['A', 'B'], columns=['C'],
   values='D', aggfunc=np.sum)
>>> table
          small  large
foo  one  1      4
     two  6      NaN
bar  one  5      4
     two  6      7
```

```
# The values arg is optional
>>> df["E"] = randn(9)
>>> df.pivot_table(index=['A', 'B'],
                   columns=['C'])
         D                E
C        large  small     large     small
A   B
bar one      4      5  1.683667 -1.979804
    two      7      6 -1.790215 -0.595985
foo one      2      1  1.256463 -0.305674
    two    NaN      3       NaN  1.172797
```