Reference

DataCamp course

Time Series Basics

**Time series are Series with special datetime type index. Therefore, many data manipulating techniques applied on pandas DataFrame or pandas Series are also applicable to time series.

Datetime without pandas

In [9]:
```
from datetime import datetime
start_target = datetime(year=2017, month=1, day=1)
print(start_target)
```

```
2017-01-01 00:00:00
```

Creating and using a Datetime Index

In [10]:
```python
import pandas as pd
date_list = ['2010-12-31 13:00:00','2010-12-31 14:00:00','2010-12-31 15:00:00','2010-12-31 16:00:00','2010-12-31
'2010-12-31 18:00:00']
temperature_list = [56.9,58.2,58.8,57.6,54.3,53.9]

#The following format will give the same results.
#time_format='%Y-%m-%d %H:%M:%S'
time_format='%Y-%m-%d'
#time_format='%Y-%m-%d %H:%M:%S'

# Convert date_list into a datetime object: my_datetimes
my_datetimes = pd.to_datetime(date_list, format=time_format)

time_series = pd.Series(temperature_list, index=my_datetimes)
# time series in a Series object with its index as datatime object.
print(time_series)
print(type(time_series))
```

```
2010-12-31 13:00:00    56.9
2010-12-31 14:00:00    58.2
2010-12-31 15:00:00    58.8
2010-12-31 16:00:00    57.6
2010-12-31 17:00:00    54.3
2010-12-31 18:00:00    53.9
dtype: float64
<class 'pandas.core.series.Series'>
```

Partial string indexing and slicing

Time series is just a series with special index date types. So indexing and slicing are straightforward as other index. The following three ways give exactly the same result.

In [11]:

```
print (time_series['2010-12-31 13:00:00':'2010-12-31 16:00:00'])
print (time_series.loc['2010-12-31 13:00:00':'2010-12-31 16:00:00'])
print (time_series.iloc[0:4])
```

```
2010-12-31 13:00:00    56.9
2010-12-31 14:00:00    58.2
2010-12-31 15:00:00    58.8
2010-12-31 16:00:00    57.6
dtype: float64
2010-12-31 13:00:00    56.9
2010-12-31 14:00:00    58.2
2010-12-31 15:00:00    58.8
2010-12-31 16:00:00    57.6
dtype: float64
2010-12-31 13:00:00    56.9
2010-12-31 14:00:00    58.2
2010-12-31 15:00:00    58.8
2010-12-31 16:00:00    57.6
dtype: float64
```

Reindexing the Index

In [12]:

```python
import pandas as pd
import numpy as np
ts1_date_list = ['2016-07-01','2016-07-02','2016-07-03','2016-07-04','2016-07-05','2016-07-06','2016-07-07','2016
                 '2016-07-09','2016-07-10','2016-07-11','2016-07-12','2016-07-13','2016-07-14','2016-07-15','2016
                 '2016-07-17']
ts2_date_list =['2016-07-01','2016-07-04','2016-07-05','2016-07-06','2016-07-07','2016-07-08','2016-07-11','2016-
                '2016-07-13','2016-07-14','2016-07-15']

ts1_list = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
ts2_list = [0,1,2,3,4,5,6,7,8,9,10]

ts1_datetimes = pd.to_datetime(ts1_date_list, format='%Y-%m-%d')
ts1 = pd.Series(ts1_list, index =ts1_datetimes)

ts2_datetimes = pd.to_datetime(ts2_date_list, format='%Y-%m-%d')
ts2 = pd.Series(ts2_list, index =ts2_datetimes)
#Above, my code

ts3 = ts2.reindex(ts1.index)

# Reindex with fill method, using forward fill: ts4. Without forward fill, NaN will be filled.
ts4 = ts2.reindex(ts1.index, method='ffill')

sum12 = ts1 + ts2
sum13 = ts1 + ts3
sum14 = ts1 + ts4

print(sum14)
```

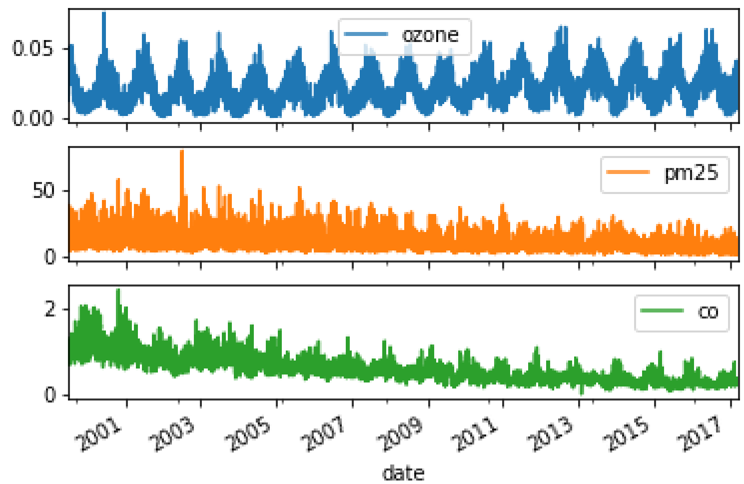
```
2016-07-01     0
2016-07-02     1
2016-07-03     2
2016-07-04     4
2016-07-05     6
2016-07-06     8
2016-07-07    10
2016-07-08    12
2016-07-09    13
2016-07-10    14
2016-07-11    16
2016-07-12    18
2016-07-13    20
```

```
2016-07-14    22
2016-07-15    24
2016-07-16    25
2016-07-17    26
dtype: int64
```

In [13]:
```
seven_days = pd.date_range('2017-1-1', periods=7)
for day in seven_days:
    print(day.dayofweek, day.weekday_name)
```

```
6 Sunday
0 Monday
1 Tuesday
2 Wednesday
3 Thursday
4 Friday
5 Saturday

C:\Users\ljyan\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: FutureWarning: `weekday_name` is deprecated and will be removed in a future version. Use `day_name` instead
  This is separate from the ipykernel package so we can avoid doing imports until
```

In [14]:
```
import matplotlib.pyplot as plt
import pandas as pd

#The following is one way, summarize other ways and put them in one place.
data = pd.read_csv('nyc.csv')
data.date = pd.to_datetime(data.date)
data.set_index('date', inplace=True) #Something new, inplace = True. Summarize with other set_index

data.plot(subplots=True)
plt.show()
```



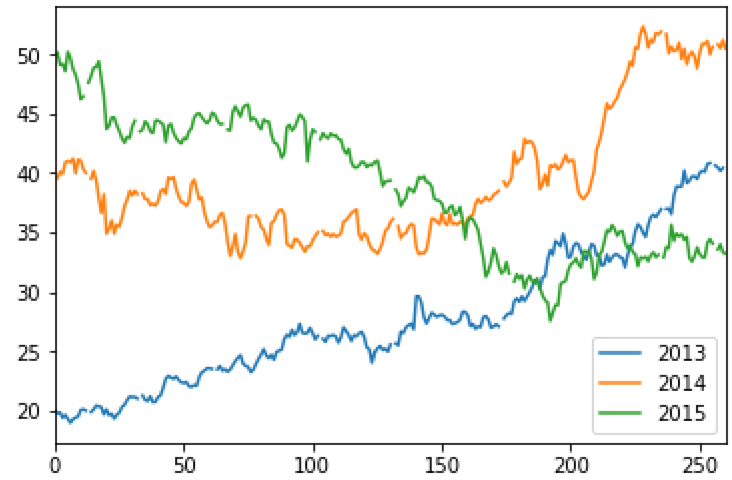
Compare annual stock price trends

Here single column yahoo prices for different years are changed to multi-year column prices. This is similar to the manipulation learned before. Connect them in the future.

Try using pandas .pivot() to implement the same function. However the following way of using .concat() is straightforward. From the original dataframe, extract three time series, and then concatenate.

In [15]:

```
import matplotlib.pyplot as plt
import pandas as pd
yahoo = pd.read_csv('yahoo.csv', index_col = 'date', parse_dates = True)
#If not set index to be datetime type, then the yahoo.loc[year,...] will not resolve 'year'

prices = pd.DataFrame()
# An empty DataFrame

for year in ['2013', '2014', '2015']:
    price_per_year = yahoo.loc[year, ['price']].reset_index(drop=True)
    #It is not always necessary to reset_index before .concat(). See example in the later cells.
    #reset_index(drop = True) removes the DatetimeIndex. Check the result if without this part.
    #Also note that even after droping DatetimeIndex, there is still default index left.

    price_per_year.rename(columns={'price': year}, inplace=True)
    #print((price_per_year))
    #Rename column name on site.

    prices = pd.concat([prices, price_per_year], axis=1)
    #Must be very clear with this part. The critical part for transformation in this problem.
    #The number of rows for each price_per_year (for 2013, 2014, 2015) might be different. See how they
    #handle this when concatecating.

# Plot prices

prices.plot()
plt.show();
```

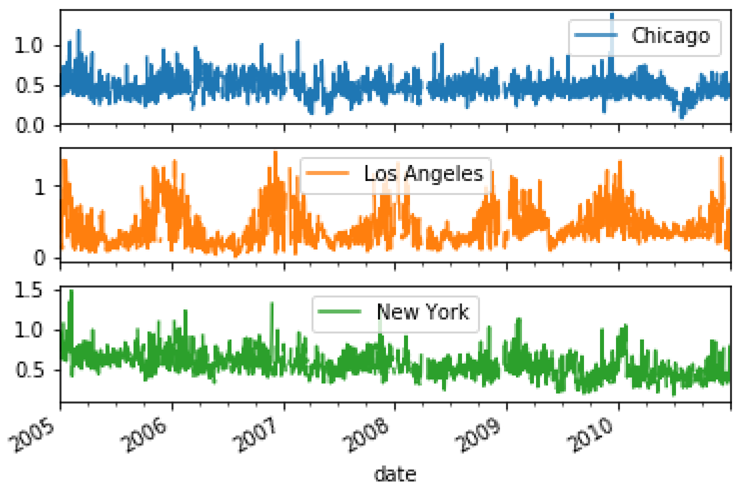


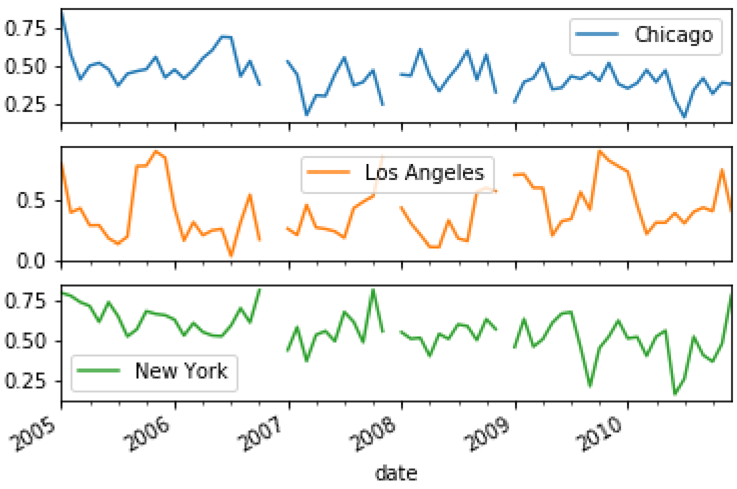
Set and change time series frequency

This should be different from resampling, where aggregation is usually made.

In [16]:

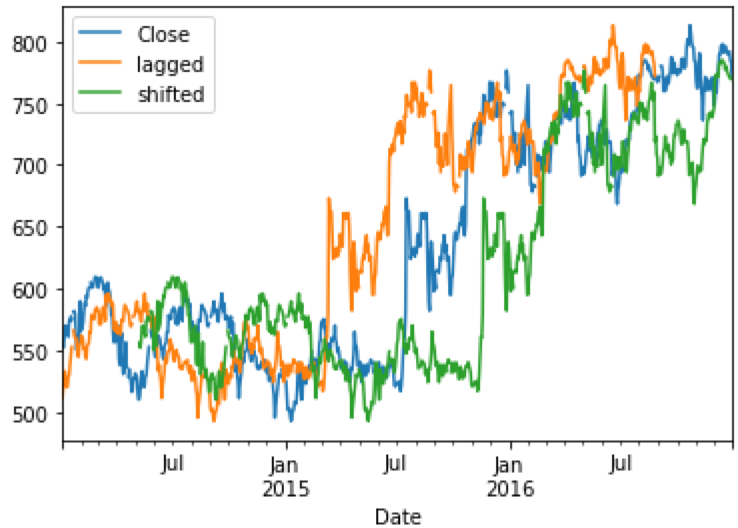
```
import pandas as pd
co = pd.read_csv('co_cities.csv',index_col = 'date', parse_dates = True)

# set the frequency to calendar daily
co = co.asfreq('D')
co.plot(subplots=True)
plt.show()
# Check other ways learned before to do the similar things.

# Set frequency to monthly
co = co.asfreq('M')
co.plot(subplots=True)
plt.show()
```





Shifting stock prices across time

In [17]:
```
google = pd.read_csv('google.csv', parse_dates=['Date'], index_col='Date') #parse_dates= ['Date']

# Set data frequency to business daily. This is something new.
google = google.asfreq('B')

# Create 'lagged' and 'shifted'. Something new. Other ways of shifting?
google['lagged'] = google.Close.shift(periods=-90)
google['shifted'] = google.Close.shift(periods=90)

google.plot()
plt.show()
```



Calculating stock price changes

Return can be calculated from different shifts.

When do we have to use yahoo.price instead of yahoo['price']? These two versions give the same results.

In [18]:
```python
yahoo['shifted_30'] = yahoo.price.shift(30)
#yahoo['shifted_30'] = yahoo['price'].shift(30)

# Subtract shifted_30 from price
yahoo['change_30'] = yahoo.price.sub(yahoo.shifted_30)

# Get the 30-day price difference
yahoo['diff_30'] = yahoo.price.diff(30)

# Inspect the last five rows of price
print(yahoo.tail())

# Show the value_counts of the difference between change_30 and diff_30
print(yahoo.change_30.sub(yahoo.diff_30).value_counts())
```

```
            price  shifted_30  change_30  diff_30
date
2015-12-25    NaN       32.19        NaN      NaN
2015-12-28  33.60       32.94       0.66     0.66
2015-12-29  34.04       32.86       1.18     1.18
2015-12-30  33.37       32.98       0.39     0.39
2015-12-31  33.26       32.62       0.64     0.64
0.0    703
dtype: int64
```

Resampling and aggregating

Resampling, Comparing different time series by normalizing their start points.

Resampling and frequency

**It is just like 'group by' with aggregation function in SQL except here the subgroup is 'more regular'.

In [19]:
```python
import pandas as pd
df = pd.read_csv('weather_data_austin_2010.csv', index_col='Date', parse_dates=True)
#In the data frame, 'Date' column is not the first column.
#with parse_dates = True, the index will automatically in datetime type.
print(df.head())

# Downsample to 6 hour data and aggregate by mean: df1
df1 = df['Temperature'].resample('6h').mean()

# Downsample to daily data and count the number of data points: df2
df2 = df['Temperature'].resample('D').count()
#print(df2)
```

```
                     Temperature  DewPoint  Pressure
Date
2010-01-01 00:00:00         46.2      37.5       1.0
2010-01-01 01:00:00         44.6      37.1       1.0
2010-01-01 02:00:00         44.1      36.9       1.0
2010-01-01 03:00:00         43.8      36.9       1.0
2010-01-01 04:00:00         43.5      36.8       1.0
```

Separating and resampling

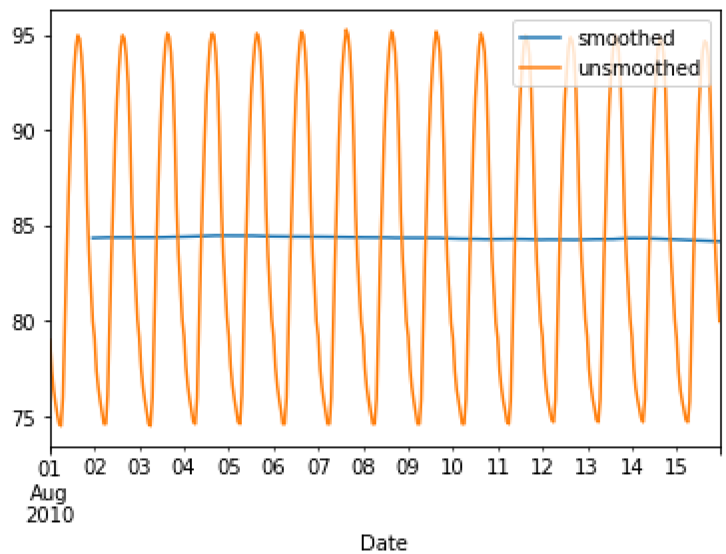
In [20]:
```python
import pandas as pd
df = pd.read_csv('weather_data_austin_2010.csv', index_col='Date', parse_dates=True)
# print(df.head())
august = df['Temperature']['2010-August']
august_highs = august.resample('D').max()
february = df['Temperature']['2010-Feb']
february_lows = february.resample('D').min()
# print(august_highs)
```

Rolling mean and frequency

Rolling means (or moving averages) are generally used to smooth out short-term fluctuations in time series data and highlight long-term trends. To use the .rolling() method, you must always use method chaining, first calling .rolling() and then chaining an aggregation method after it.

Comments: Like resampling, rolling is also related to 'group by' + aggregate function except the grouping scheme is different.

In [21]:
```python
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('weather_data_austin_2010.csv', index_col='Date', parse_dates=True)
#print(df.head())

unsmoothed = df.loc['2010-Aug-01':'2010-Aug-15','Temperature']
#unsmoothed.head()

smoothed = unsmoothed.rolling(window=24).mean()

august = pd.DataFrame({'smoothed':smoothed, 'unsmoothed':unsmoothed})

august.plot()
plt.show()
print(len(unsmoothed),len(smoothed))
#Althought the length of unsmoothed and smoothed are same, the first 24-1 entries of smoothed is NaN.
# print(smoothed.head(5))
```




```
360 360
```

Resample and roll with it

In [22]:
```
import pandas as pd
df = pd.read_csv('weather_data_austin_2010.csv', index_col='Date', parse_dates=True)
august = df.loc['2010-Aug','Temperature'] #The old version from Datacamp does not work here. So I changed to loc.
daily_highs = august.resample('D').max()
daily_highs_smoothed = august.resample('D').max().rolling(window=7).mean()
print(daily_highs_smoothed.head(10))
```

```
Date
2010-08-01          NaN
2010-08-02          NaN
2010-08-03          NaN
2010-08-04          NaN
2010-08-05          NaN
2010-08-06          NaN
2010-08-07    95.114286
2010-08-08    95.142857
2010-08-09    95.171429
2010-08-10    95.171429
Freq: D, Name: Temperature, dtype: float64
```

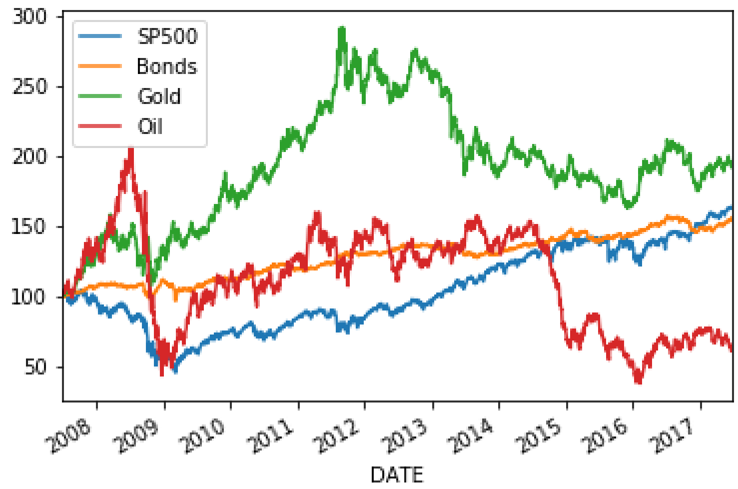
Method chaining and filtering

In [23]:

```python
import pandas as pd
df = pd.read_csv("austin_airport_departure_data_2015_july.csv", index_col='Date (MM/DD/YYYY)', parse_dates=True)
#print(df.head())

# Strip extra whitespace from the column names: df.columns
# It is not called whitespace when they are between two words. Here it refers to the
# spaces in the end of column names. This is a very special case harder to find.
df.columns = df.columns.str.strip()
#print(df.head())

# Extract data for which the destination airport is Dallas: dallas
dallas = df['Destination Airport'].str.contains('DAL')
# This is just a specal boolear mask. It is just like other filter function that gives a boolean mask.

daily_departures = dallas.resample('D').sum()

print(daily_departures.head())
stats = daily_departures.describe()
print(stats)
```

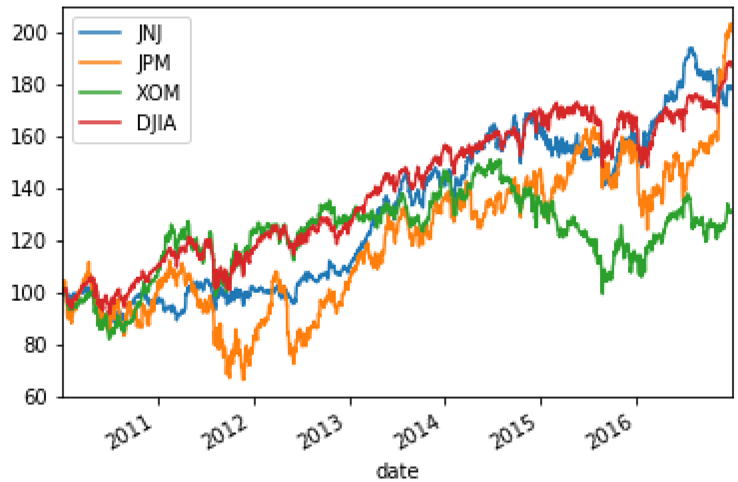
```
Date (MM/DD/YYYY)
2015-07-01    10.0
2015-07-02    10.0
2015-07-03    11.0
2015-07-04     3.0
2015-07-05     9.0
Freq: D, Name: Destination Airport, dtype: float64
count    31.000000
mean      9.322581
std       1.989759
min       3.000000
25%       9.500000
50%      10.000000
75%      10.000000
max      11.000000
Name: Destination Airport, dtype: float64
```

In [24]:
```
prices = pd.read_csv('asset_classes.csv', parse_dates=['DATE'], index_col='DATE')
first_prices = prices.iloc[0]
normalized = prices.div(first_prices).mul(100)
normalized.plot()
plt.show()
```



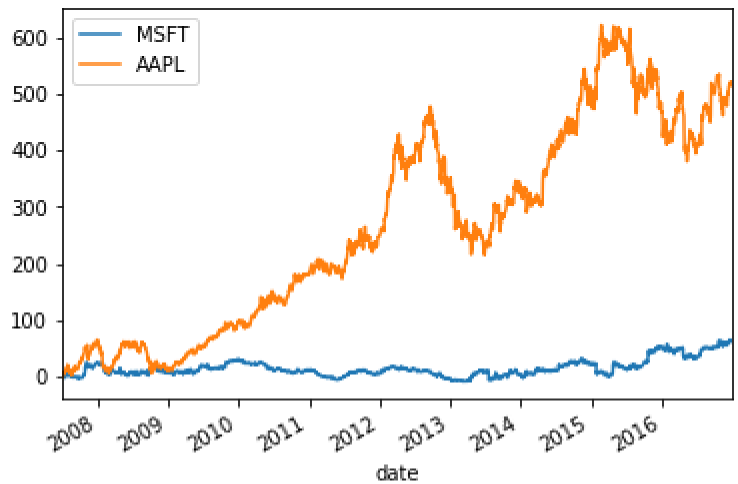
Comparing stock prices with a benchmark

In [25]:
```
stocks = pd.read_csv('nyse.csv', parse_dates=['date'], index_col='date')
dow_jones = pd.read_csv('dow_jones.csv', parse_dates=['date'], index_col='date')

data = pd.concat([stocks, dow_jones], axis=1)

data.div(data.iloc[0]).mul(100).plot()
plt.show()
```



Plot performance DIFFERENCE vs benchmark index

In [26]:
```
tickers = ['MSFT', 'AAPL']
stocks = pd.read_csv('msft_aapl.csv', parse_dates=['date'], index_col='date')
sp500 = pd.read_csv('sp500.csv', parse_dates=['date'], index_col='date')

data = pd.concat([stocks, sp500], axis=1).dropna() #drop nan.
normalized = data.div(data.iloc[0]).mul(100)

# Subtract the normalized index from the normalized stock prices, and plot the result
normalized[tickers].sub(normalized['SP500'], axis=0).plot()
plt.show()
```



Convert monthly to weekly data

In [27]:
```python
start = '2016-1-1'
end = '2016-2-29'
monthly_dates = pd.date_range(start=start, end=end, freq='M')
# This saves the manual date list creation as in other places. Try to do this for the notes elsewhere.

monthly = pd.Series(data=[1,2], index=monthly_dates)
print(monthly)

weekly_dates = pd.date_range(start=start, end=end, freq='W')

# Print monthly, reindexed using weekly_dates
print(monthly.reindex(weekly_dates))
print(monthly.reindex(weekly_dates, method='bfill'))
print(monthly.reindex(weekly_dates, method='ffill'))
```

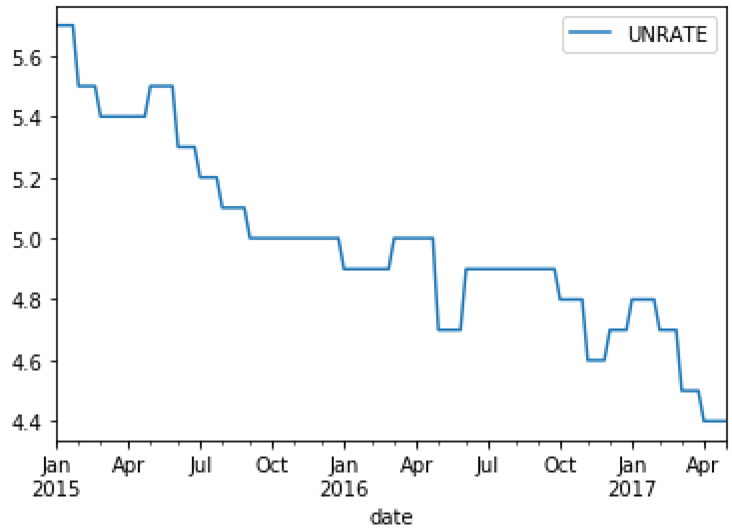
```
2016-01-31    1
2016-02-29    2
Freq: M, dtype: int64
2016-01-03    NaN
2016-01-10    NaN
2016-01-17    NaN
2016-01-24    NaN
2016-01-31    1.0
2016-02-07    NaN
2016-02-14    NaN
2016-02-21    NaN
2016-02-28    NaN
Freq: W-SUN, dtype: float64
2016-01-03    1
2016-01-10    1
2016-01-17    1
2016-01-24    1
2016-01-31    1
2016-02-07    2
2016-02-14    2
2016-02-21    2
2016-02-28    2
Freq: W-SUN, dtype: int64
2016-01-03    NaN
2016-01-10    NaN
2016-01-17    NaN
```

```
2016-01-24    NaN
2016-01-31    1.0
2016-02-07    1.0
2016-02-14    1.0
2016-02-21    1.0
2016-02-28    1.0
Freq: W-SUN, dtype: float64
```

Create weekly from monthly unemployment data

In [28]:
```python
data = pd.read_csv('unrate_2000.csv', parse_dates=['date'], index_col='date')
print(data.asfreq('W').head())
print(data.asfreq('W', method='bfill').head())
weekly_ffill = data.asfreq('W', method='ffill')
print(weekly_ffill.head())
weekly_ffill.loc['2015':].plot()
plt.show()
```

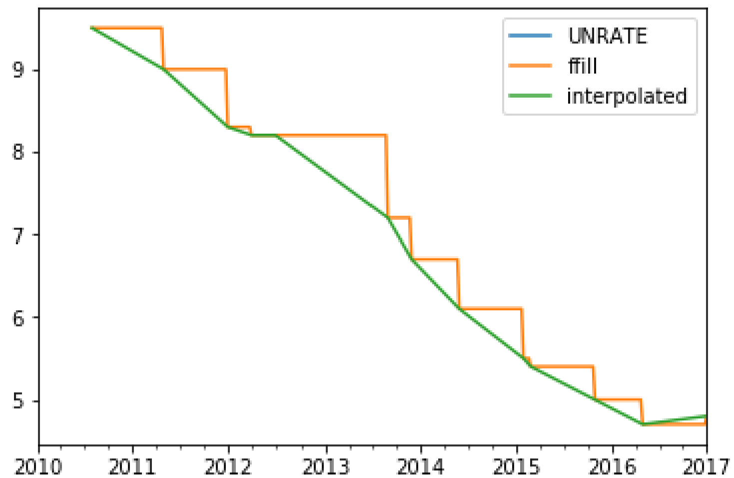
```
            UNRATE
date
2000-01-02     NaN
2000-01-09     NaN
2000-01-16     NaN
2000-01-23     NaN
2000-01-30     NaN
            UNRATE
date
2000-01-02     4.1
2000-01-09     4.1
2000-01-16     4.1
2000-01-23     4.1
2000-01-30     4.1
            UNRATE
date
2000-01-02     4.0
2000-01-09     4.0
2000-01-16     4.0
2000-01-23     4.0
2000-01-30     4.0
```



Use interpolation to create weekly employment data

In [29]:
```
monthly = pd.read_csv('unrate.csv', parse_dates=['DATE'], index_col='DATE')

print(monthly.info())
weekly_dates = pd.date_range(monthly.index.min(), monthly.index.max(), freq='W')
#does not need to hard-coding
weekly = monthly.reindex(weekly_dates)
weekly['ffill'] = weekly.UNRATE.ffill()
weekly['interpolated'] = weekly.UNRATE.interpolate()
weekly.plot()
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 85 entries, 2010-01-01 to 2017-01-01
Data columns (total 1 columns):
UNRATE    85 non-null float64
dtypes: float64(1)
memory usage: 1.3 KB
None
```

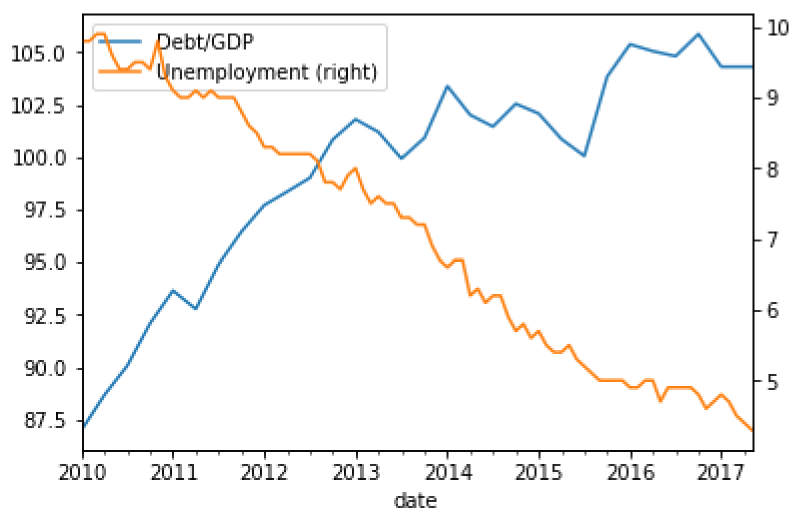


Interpolate debt/GDP and compare to unemployment

In [30]:

```
data = pd.read_csv('debt_unemployment.csv', parse_dates=['date'], index_col='date')
print(data.info())

interpolated = data.interpolate()
print(interpolated.info())

interpolated.plot(secondary_y='Unemployment');
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 89 entries, 2010-01-01 to 2017-05-01
Data columns (total 2 columns):
Debt/GDP        29 non-null float64
Unemployment    89 non-null float64
dtypes: float64(2)
memory usage: 2.1 KB
None
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 89 entries, 2010-01-01 to 2017-05-01
Data columns (total 2 columns):
Debt/GDP        89 non-null float64
Unemployment    89 non-null float64
dtypes: float64(2)
memory usage: 2.1 KB
None
```



Missing values and interpolation

This is usually used in stock time series where data are missing

In [31]:
```python
import pandas as pd
import numpy as np
ts1_date_list = ['2016-07-01','2016-07-02','2016-07-03','2016-07-04','2016-07-05','2016-07-06','2016-07-07','2016
                 '2016-07-09','2016-07-10','2016-07-11','2016-07-12','2016-07-13','2016-07-14','2016-07-15','2016
                 '2016-07-17']
ts2_date_list =['2016-07-01','2016-07-04','2016-07-05','2016-07-06','2016-07-07','2016-07-08','2016-07-11','2016-
                '2016-07-13','2016-07-14','2016-07-15']

ts1_list = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
ts2_list = [0,1,2,3,4,5,6,7,8,9,10]

ts1_datetimes = pd.to_datetime(ts1_date_list, format='%Y-%m-%d')
ts1 = pd.Series(ts1_list, index =ts1_datetimes)

ts2_datetimes = pd.to_datetime(ts2_date_list, format='%Y-%m-%d')
ts2 = pd.Series(ts2_list, index =ts2_datetimes)
#Above my code

ts2_interp = ts2.reindex(ts1.index).interpolate(how='linear')
print(ts2_interp)
differences = np.abs(ts1 - ts2_interp)
print(differences.describe())
```

```
2016-07-01     0.000000
2016-07-02     0.333333
2016-07-03     0.666667
2016-07-04     1.000000
2016-07-05     2.000000
2016-07-06     3.000000
2016-07-07     4.000000
2016-07-08     5.000000
2016-07-09     5.333333
2016-07-10     5.666667
2016-07-11     6.000000
2016-07-12     7.000000
2016-07-13     8.000000
2016-07-14     9.000000
2016-07-15    10.000000
2016-07-16    10.000000
2016-07-17    10.000000
dtype: float64
count    17.000000
```

```
mean      2.882353
std       1.585267
min       0.000000
25%       2.000000
50%       2.666667
75%       4.000000
max       6.000000
dtype: float64
```

Time zones and conversion

Time zone handling with **pandas typically assumes that you are handling the Index of the Series**. Here however, we **handle timezones that are associated with datetimes in the column data**.
Note there are some weird white space in column names, e.g. at the end of 't' of 'Destination Airport '. So I need get it stripped before going on. Otherwise, it will cause weird problems.

In [32]:

```
import pandas as pd
df = pd.read_csv("austin_airport_departure_data_2015_july.csv")
#If I read in the Date column as index, then it is convenient for slicing,
#but I cannot add the time strings from two columns.

df.columns = df.columns.str.strip()

mask = df['Destination Airport'] == 'LAX'

la = df[mask]

L_A = df['Destination Airport'].str.contains('LAX')
#print(L_A)
#Here L_A is equivalent to the mask above, but not equivalent to la.

times_tz_none = pd.to_datetime( la['Date (MM/DD/YYYY)'] + ' ' + la['Wheels-off Time'] )
# to_datetime transform a string object to datetime object.
# times_tz_none is zero-zone? Otherwise, there will be a problem.

# Localize the time to US/Central: times_tz_central
times_tz_central = times_tz_none.dt.tz_localize('US/Central')

# Convert the datetimes from US/Central to US/Pacific
times_tz_pacific = times_tz_central.dt.tz_convert('US/Pacific')
```

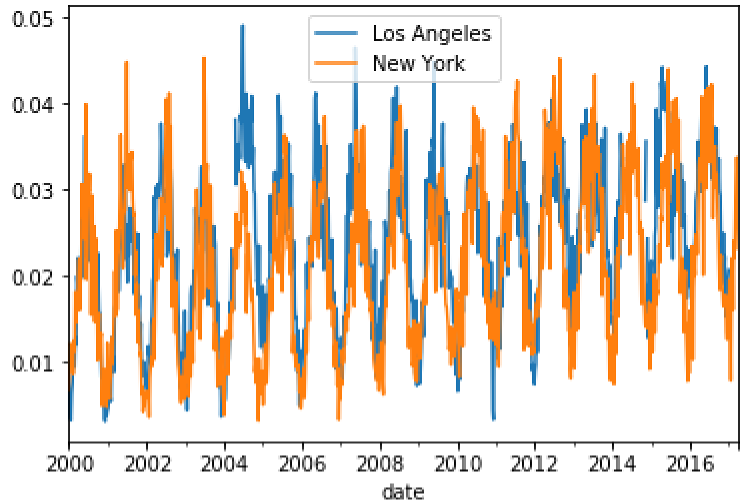
Dropping rows with datetime index

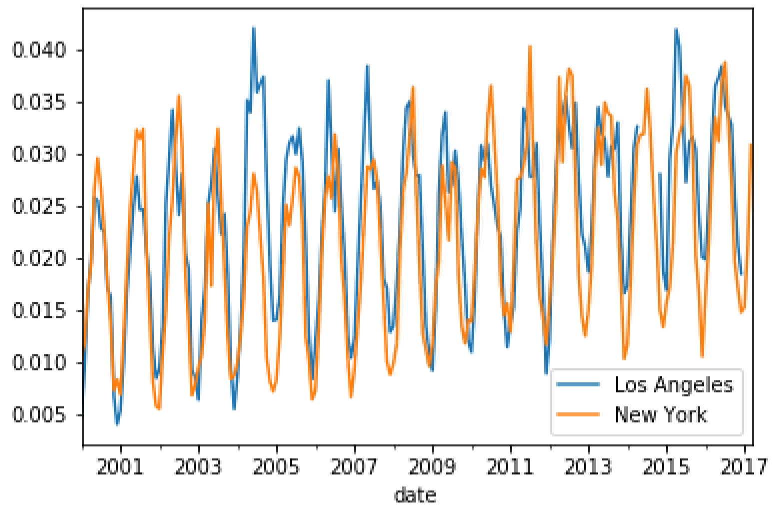
In [ ]:
```
returns = returns.drop(pd.Timestamp('2012-08-12'))
#This is for data frame. If returns is a Series type, then the grammar is different. Check Google.
#It might be possible to drop just 0 row index.

returns = returns.dropna()
#This is OK only for dropping NaN entries.
```

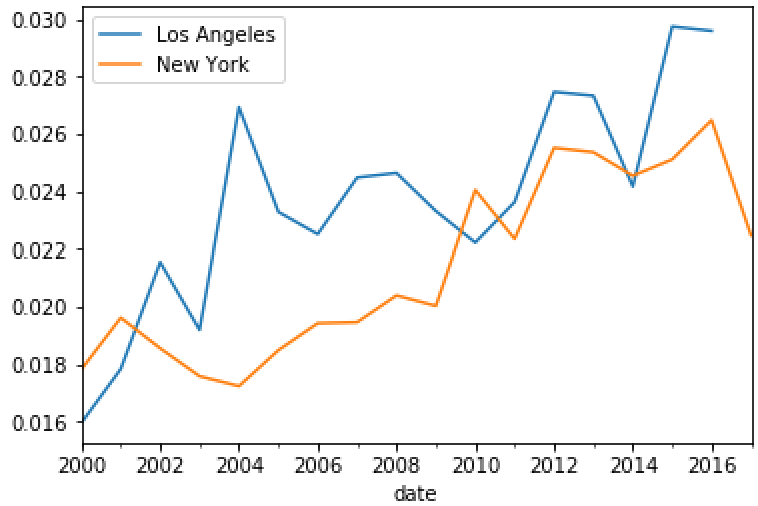
Compare weekly, monthly and annual ozone trends for NYC & LA

Be clear what is downsampling and upsampling.

In [34]:
```
ozone = pd.read_csv('ozone_nyla.csv', parse_dates=['date'], index_col='date')
ozone.resample('W').mean().plot();
plt.show()

ozone.resample('M').mean().plot();
plt.show();

ozone.resample('A').mean().plot();
plt.show();
```







Compare quarterly GDP growth rate and stock returns

With the skill to downsample and aggregate time series, we can compare higher-frequency stock price series to lower-frequency economic time series.

In [35]:
```
gdp_growth = pd.read_csv('gdp_growth.csv', parse_dates=['date'], index_col='date')

djia = pd.read_csv('djia.csv', parse_dates=['date'], index_col='date')

# Calculate djia quarterly returns here
# GDP growth is reported at the beginning of each quarter for the previous quarter. To calculate matching stock r
# you'll resample the stock index to quarter start frequency using the alias 'QS', and aggregating using the .fir
# observations.
djia_quarterly = djia.resample('QS').first()
djia_quarterly_return = djia_quarterly.pct_change().mul(100)

# Concatenate, rename and plot djia_quarterly_return and gdp_growth here
data = pd.concat([gdp_growth, djia_quarterly_return], axis=1)
data.columns = ['gdp', 'djia']

data.plot()
plt.show();
```

Visualize monthly mean, median and standard deviation of S&P500 returns

In [36]:
```
sp500 = pd.read_csv('sp500.csv', parse_dates=['date'], index_col='date')

# Calculate daily returns here,it is pct_change()'s default. If other returns, need specify parameters.
daily_returns = sp500.squeeze().pct_change()
#squeeze() returns scalar if 1-sized, else original object

stats = daily_returns.resample('M').agg(['mean', 'median', 'std'])

stats.plot()
plt.show();
```

Window Functions: Rolling & Expanding Metrics

[https://www.tutorialspoint.com/python_pandas/python_pandas_window_functions.htm (https://www.tutorialspoint.com/python_pandas/python_pandas_window_functions.htm)](https://www.tutorialspoint.com/python_pandas/python_pandas_window_functions.htm) For working on numerical data, Pandas provide few variants like rolling, expanding and exponentially moving weights for window statistics. Among these are sum, mean, median, variance, covariance, correlation, etc. Window functions are usually used in finding the trends within the data graphically by smoothing the curve. Check pandas.rolling() and pandas.expanding() for specific examples. The accumulative sum/product methods in this chapter sometimes are equivalent to the .expanding().

The .expanding() is actually very useful, because rolling operations are for **fixed window sizes**. However, this operation is an expanding window size. It starts with a "rolling" window of length 1 period, the next window size is 2 periods, then 3, 4, 5, etc. For streaming data a rolling window of the full length of the original dataframe will start to drop the first couple of observations, whereas the expanding window allows you to add new data.

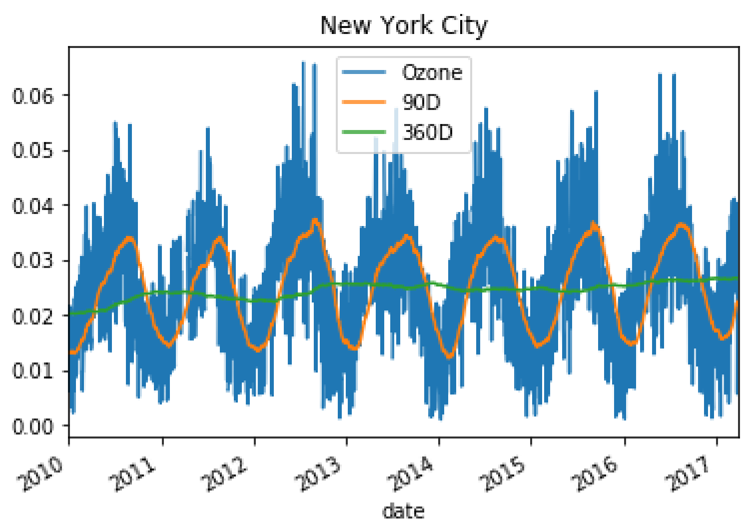
See the pandas functions (defined on dataframe) cummax, cummin, cumprod,...which have similar expanding behavior described above.

Finally, window functions above are to some extent similar to the group by statement in SQL: find subgroups and find aggregation statistics on each subgroup. However, SQL also provides window functions. Compare the window functions here and those in T-SQL.

Rolling average air quality since 2010 for new york city

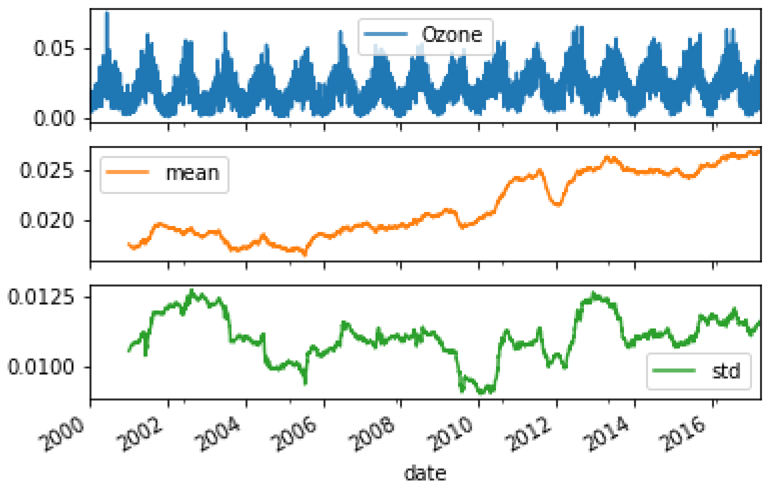
In [37]:
```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('ozone_nyc.csv', parse_dates=['date'], index_col='date')
print(data.info())

data['90D'] = data.Ozone.rolling('90D').mean()
data['360D'] = data.Ozone.rolling('360D').mean()

data['2010':].plot(title='New York City')
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6291 entries, 2000-01-01 to 2017-03-31
Data columns (total 1 columns):
Ozone    6167 non-null float64
dtypes: float64(1)
memory usage: 98.3 KB
None
```



Rolling 360-day median & std. deviation for nyc ozone data since 2000

In [38]:
```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('ozone_nyc.csv', parse_dates=['date'], index_col='date').dropna()

rolling_stats = data.Ozone.rolling(360).agg(['mean', 'std'])

# Join rolling_stats with ozone data
stats = data.join(rolling_stats)

stats.plot(subplots=True);
plt.show()
```



Rolling quantiles for daily air quality in nyc

My understanding of following example: from the first 1-360 data points, calculate the 0.1, 0.5,0.9 quantile. Then from 2-361 data points calculate another set of 0.1, 0.5, 0.9 quantiles,....

0 quartile = 0 quantile = 0 percentile

1 quartile = 0.25 quantile = 25 percentile

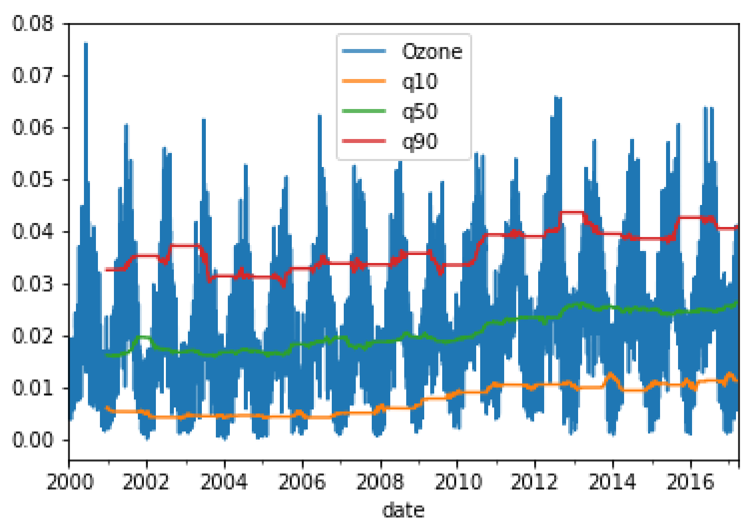
2 quartile = .5 quantile = 50 percentile (median)

3 quartile = .75 quantile = 75 percentile

4 quartile = 1 quantile = 100 percentile

In [39]:
```
data = pd.read_csv('ozone_nyc.csv', parse_dates=['date'], index_col='date').dropna()

data = data.resample('D').interpolate()
data.info()
rolling = data.rolling(360)['Ozone']
print(type(rolling)) #what is the structure of a rolling object, particularly as compared to the original DataFra

# Insert the rolling quantiles to the monthly returns
data['q10'] = rolling.quantile(.1)
data['q50'] = rolling.quantile(.5)
data['q90'] = rolling.quantile(.9)

data.plot()
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6300 entries, 2000-01-01 to 2017-03-31
Freq: D
Data columns (total 1 columns):
Ozone    6300 non-null float64
dtypes: float64(1)
memory usage: 98.4 KB
<class 'pandas.core.window.Rolling'>
```



Cumulative sum vs .diff()

The cumulative sum method is similar to the pandas.expanding() introduced earlier. It can be taken as another way of forming groups to apply aggregation. .expanding is different from .rolling (see note before).

The .diff() and .cumsum() are opposite. It is easy to find an example to show this.

The cumulative sum can be used to calculate the return within the former, 1 day, 1 week, 1 month etc.

Related cumsum() to CDF.

In [40]:
```
data = pd.read_csv('google.csv', index_col = 'Date', parse_dates = True)
data = data.dropna()

differences = data.diff().dropna()
#if diff() has no parameter, then it should be shifted one unit.

# Select start price
start_price = data.first('D')
print(start_price)

# Calculate cumulative sum
cumulative_sum = start_price.append(differences).cumsum()

# Validate cumulative sum equals data
print(data.equals(cumulative_sum))
```

```
            Close
Date
2014-01-02  556.0
True
```

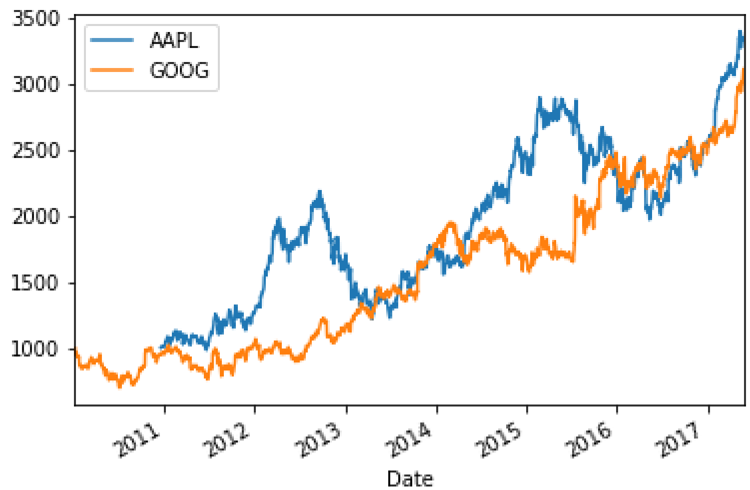
Cumulative return on 1,000 invested in google vs apple I

To calculate **cumulative return** calculations to practical use, let's compare how much $1,000 would be worth if invested in Google ('GOOG') or Apple ('AAPL') in 2010.

Some related functions defined in DataFrame. **The cumulative is usually not meaning cumulating among the whole range but cumulating in rolling approach.** For example, in the cumprod() below, it will give cumulative returns on many points but not just one in the end.

```
 DataFrame.prod
Return the product over DataFrame axis.
DataFrame.cummax
Return cumulative maximum over DataFrame axis.
DataFrame.cummin
Return cumulative minimum over DataFrame axis.
DataFrame.cumsum
Return cumulative sum over DataFrame axis.
DataFrame.cumprod
Return cumulative product over DataFrame axis.
```

In [41]:
```
data = pd.read_csv('apple_google.csv', index_col = 'Date', parse_dates = True)

investment = 1000
returns = data.pct_change()

# Calculate the cumulative returns here
returns_plus_one = returns.add(1) #Important. It is about x*(1+0.12)*(1+0.08)...

cumulative_return = returns_plus_one.cumprod()
#Cumulative_return is not just the cumulative return for the whole range, but include cumulative returns on each

# Calculate and plot the investment return here
cumulative_return.mul(investment).plot()
plt.show();
```



Cumulative return on 1,000 invested in google vs apple II

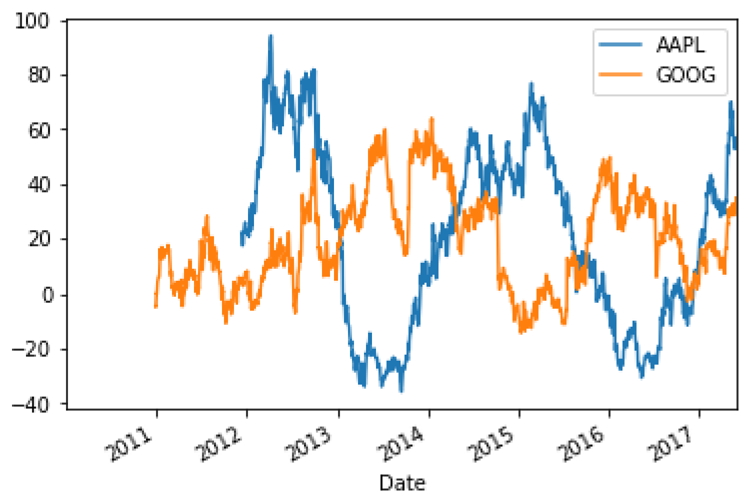
Apple outperformed Google over the entire period, but this may have been different over various 1-year sub periods, so that switching between the two stocks might have yielded an even better result.

To analyze this, calculate that cumulative return for rolling 1-year periods, and then plot the returns to see when each stock was superior.

In a rolling window, we can use the standard built in window functions such as mean, std. Or we may define our own functions.

In [42]:
```
import numpy as np

# Define a multi_period_return function
def multi_period_return(period_returns):
    return np.prod(period_returns + 1) - 1
    #Be careful of the +1 and -1 when necessary.

daily_returns = data.pct_change()

# Calculate rolling_annual_returns
rolling_annual_returns = daily_returns.rolling('360D').apply(multi_period_return)

# Plot rolling_annual_returns
rolling_annual_returns.mul(100).plot();
plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\ipykernel_launcher.py:11: FutureWarning: Currently, 'apply' passes t
he values as ndarrays to the applied function. In the future, this will change to passing it as Series objects.
You need to specify 'raw=True' to keep the current behaviour, and you can pass 'raw=False' to silence this warn
ing
  # This is added back by InteractiveShellApp.init_path()
```



Putting it all together: Building a value-weighted index

This index uses market-cap data contained in the stock exchange listings to calculate weights and 2016 stock price information.

Explore and clean company listing information

First calculate market-cap weights for these stocks.

In [43]:
```python
import pandas as pd
listings = pd.read_csv('list.csv')
print(listings.info())

listings.set_index('Stock Symbol', inplace=True)
listings.dropna(subset=['Sector'], inplace=True)
# print(listings.head())

# Select companies with IPO Year before 2019
listings = listings[listings['IPO Year'] < 2019]

print(listings.info())

print(listings.groupby('Sector').size().sort_values(ascending=False))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 360 entries, 0 to 359
Data columns (total 7 columns):
Stock Symbol             360 non-null object
Company Name             360 non-null object
Last Sale                346 non-null float64
Market Capitalization    360 non-null float64
IPO Year                 105 non-null float64
Sector                   238 non-null object
Industry                 238 non-null object
dtypes: float64(3), object(4)
memory usage: 19.8+ KB
None
<class 'pandas.core.frame.DataFrame'>
Index: 45 entries, ACU to ZDGE
Data columns (total 6 columns):
Company Name             45 non-null object
Last Sale                45 non-null float64
Market Capitalization    45 non-null float64
IPO Year                 45 non-null float64
Sector                   45 non-null object
Industry                 45 non-null object
dtypes: float64(3), object(3)
memory usage: 2.5+ KB
None
Sector
```

```
Health Care              11
Consumer Services         9
Basic Industries          8
Capital Goods             5
Technology                4
Public Utilities          3
Energy                    2
Miscellaneous             1
Finance                   1
Consumer Non-Durables     1
dtype: int64
```

Select and inspect index components

After having imported and cleaned the listings data, we then select the index components as the largest company for each sector by market capitalization.

In [44]:
```python
components = listings.groupby(['Sector'])['Market Capitalization'].nlargest(1)

print(components.sort_values(ascending=False))
tickers = components.index.get_level_values('Stock Symbol')
print(tickers)
info_cols = ['Company Name', 'Market Capitalization', 'Last Sale']
print(listings.loc[tickers, info_cols].sort_values('Market Capitalization', ascending=False))
```

```
Sector                  Stock Symbol
Public Utilities        CQP             1.104692e+10
Finance                 SEB             4.603773e+09
Basic Industries        SIM             2.123559e+09
Consumer Services       GSAT            1.931551e+09
Health Care             CRHM            6.474389e+08
Energy                  MPO             4.794015e+08
Capital Goods           LBY             3.026988e+08
Consumer Non-Durables   ROX             2.376444e+08
Technology              MJCO            1.916146e+08
Miscellaneous           AUXO            5.913104e+07
Name: Market Capitalization, dtype: float64
Index(['SIM', 'LBY', 'ROX', 'GSAT', 'MPO', 'SEB', 'CRHM', 'AUXO', 'CQP',
       'MJCO'],
      dtype='object', name='Stock Symbol')
                                   Company Name  Market Capitalization  \
Stock Symbol
CQP                  Cheniere Energy Partners, LP           1.104692e+10
SEB                          Seaboard Corporation           4.603773e+09
SIM                       Grupo Simec, S.A. de C.V.           2.123559e+09
GSAT                             Globalstar, Inc.           1.931551e+09
CRHM                       CRH Medical Corporation           6.474389e+08
MPO              MIDSTATES PETROLEUM COMPANY, INC.           4.794015e+08
LBY                                  Libbey, Inc.           3.026988e+08
ROX                           Castle Brands, Inc.           2.376444e+08
MJCO                                      Majesco           1.916146e+08
AUXO                               Auxilio, Inc.           5.913104e+07

              Last Sale
Stock Symbol
CQP             32.7000
SEB           3933.0000
SIM             12.8000
GSAT             1.7300
```

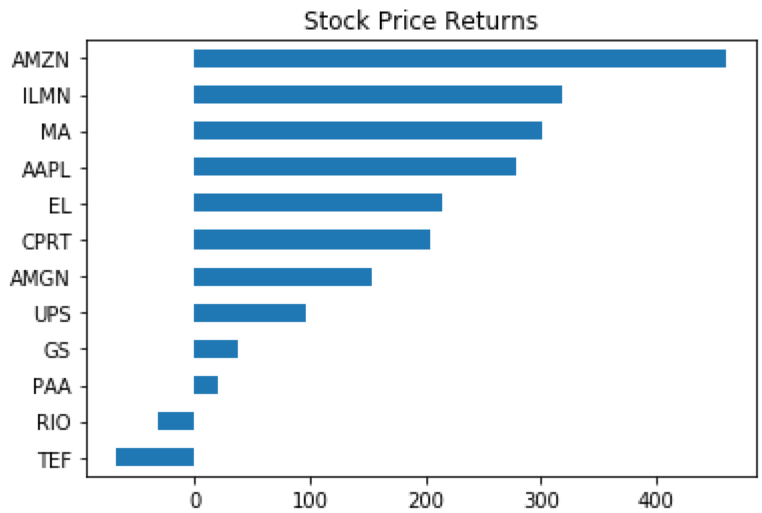
```
CRHM             8.9000
MPO             19.1800
LBY             13.8200
ROX              1.4600
MJCO             5.2500
AUXO             6.3043
```

Import index component price information

Use the stock symbols for the companies selected in the last exercise to calculate returns for each company.

In [45]:
```
import matplotlib.pyplot as plt
#The following is copied from Datacamp Ipython. It is different from the tickers calculated above. Figure out why
#The list.csv might have probles. Because the stock_data.csv only have prices for the following tickers (from Dat
#I instead use the tickers below rather than the calcualted in the previous cell
tickers = ['RIO', 'ILMN', 'CPRT', 'EL', 'AMZN', 'PAA', 'GS', 'AMGN', 'MA', 'TEF', 'AAPL', 'UPS']

# Print tickers
print(tickers)

# Import prices and inspect result
stock_prices = pd.read_csv('stock_data.csv', parse_dates=['Date'], index_col='Date')
# print(stock_prices.info())
print(stock_prices.head())

# Calculate the returns
price_return = stock_prices.iloc[-1].div(stock_prices.iloc[0]).sub(1).mul(100)

# Plot horizontal bar chart of sorted price_return
price_return.sort_values().plot(kind='barh', title='Stock Price Returns')
plt.show()
```

```
['RIO', 'ILMN', 'CPRT', 'EL', 'AMZN', 'PAA', 'GS', 'AMGN', 'MA', 'TEF', 'AAPL', 'UPS']
             AAPL   AMGN    AMZN  CPRT     EL      GS   ILMN     MA    PAA  \
Date
2010-01-04  30.57  57.72  133.90  4.55  24.27  173.08  30.55  25.68  27.00
2010-01-05  30.63  57.22  134.69  4.55  24.18  176.14  30.35  25.61  27.30
2010-01-06  30.14  56.79  132.25  4.53  24.25  174.26  32.22  25.56  27.29
2010-01-07  30.08  56.27  130.00  4.50  24.56  177.67  32.77  25.39  26.96
2010-01-08  30.28  56.77  133.52  4.52  24.66  174.31  33.15  25.40  27.05

              RIO    TEF    UPS
Date
2010-01-04  56.03  28.55  58.18
2010-01-05  56.90  28.53  58.28
2010-01-06  58.64  28.23  57.85
2010-01-07  58.65  27.75  57.41
2010-01-08  59.30  27.57  60.17
```



Calculate number of shares outstanding

Calculate the number of shares for each index component. The number of shares will allow you to calculate the total market capitalization for each component given the historical price series in the next exercise.

In [46]:
```
import pandas as pd
listings = pd.read_csv('list.csv')

listings.set_index('Stock Symbol', inplace=True)
listings.dropna(subset=['Sector'], inplace=True)

# Select companies with IPO Year before 2019
listings = listings[listings['IPO Year'] < 2019]
# Extra code above

# Inspect listings and print tickers
tickers = ['SIM', 'LBY', 'ROX', 'GSAT', 'MPO', 'SEB', 'CRHM', 'AUXO', 'CQP', 'MJCO']
#Here I use the stickers obtained from the 'wrong' list.csv. This is different from that of Datacamp.
#If I use the datacamp version, ['RIO', 'ILMN', 'CPRT', 'EL', 'AMZN', 'PAA', 'GS', 'AMGN', 'MA', 'TEF', 'AAPL',
#then the sticks will not in the listings. In other words, the sentence below will not work.

components = listings.loc[tickers, ['Market Capitalization', 'Last Sale']]

# Print the first rows of components
# print(components.head())

# Calculate the number of shares here
no_shares = components['Market Capitalization'].div(components['Last Sale'])

# Print the sorted no_shares
# print(no_shares.sort_values(ascending=False))
```

Create time series of market value

Use the number of shares to calculate the total market capitalization for each component and trading date from the historical price series.

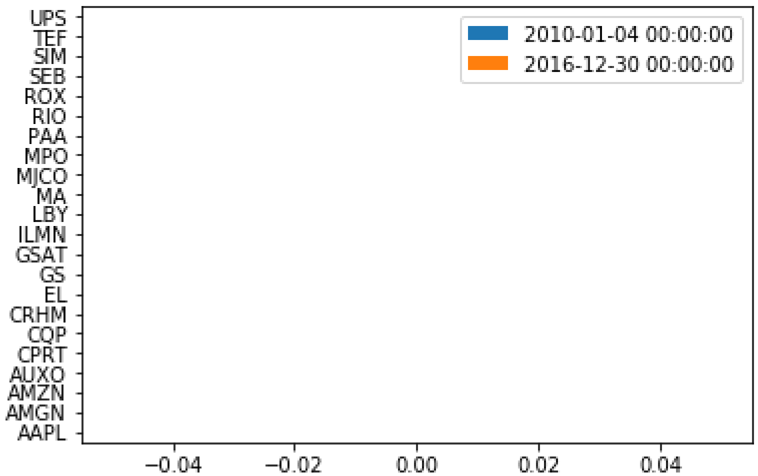
The result will be the key input to construct the value-weighted stock index.

There is no right data, so see the output of the Datacamp after the code.

In [47]:

```python
# Select the number of shares
components.head()
components['Number of Shares'] = components['Market Capitalization'].div(components['Last Sale'])
# extra code above

no_shares = components['Number of Shares']
print(no_shares.sort_values())

# Create the series of market cap per ticker
market_cap = stock_prices.mul(no_shares)
#Consider the contents of no_shares and stock_price, how they are multiplied together?

# Select first and last market cap here
first_value = market_cap.iloc[0]
last_value = market_cap.iloc[-1]

# Concatenate and plot first and last market cap here
pd.concat([first_value, last_value], axis=1).plot(kind='barh')
plt.show()
```

```
Stock Symbol
SEB     1.170550e+06
AUXO    9.379477e+06
LBY     2.190295e+07
MPO     2.499487e+07
MJCO    3.649803e+07
CRHM    7.274594e+07
ROX     1.627702e+08
SIM     1.659031e+08
CQP     3.378264e+08
GSAT    1.116503e+09
Name: Number of Shares, dtype: float64
```



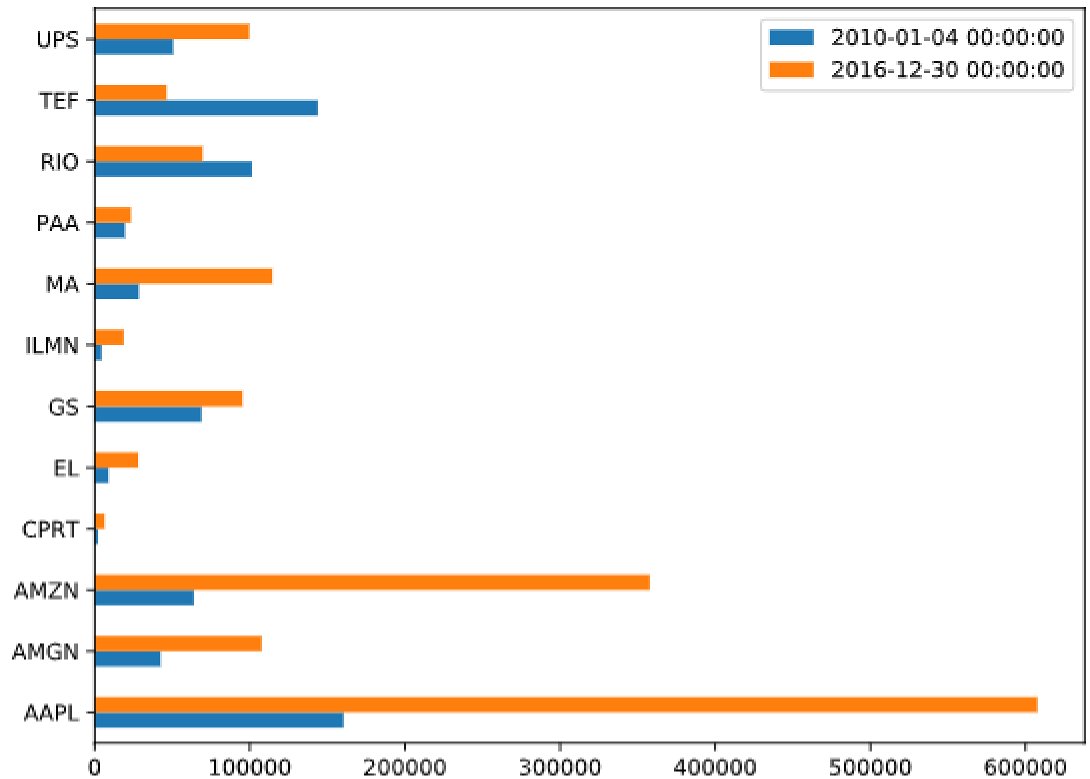
output of print(no_shares.sort_values()):

```
 Stock Symbol
ILMN      146.300000
EL        366.405816
GS        397.817439
CPRT      459.390316
AMZN      477.170618
PAA       723.404994
AMGN      735.890171
UPS       869.297154
MA       1108.884100
RIO      1808.717948
TEF      5037.804990
AAPL     5246.540000
```

Contents of stock_price:

```
 AAPL     AMGN     AMZN    CPRT      EL       GS      ILMN       MA      PAA      RIO      TEF      UPS
Date
2010-01-04    30.57    57.72  133.90    4.55  24.27  173.08    30.55    25.68  27.00  56.03  28.55    58.18
2010-01-05    30.63    57.22  134.69    4.55  24.18  176.14    30.35    25.61  27.30  56.90  28.53    58.28
2010-01-06    30.14    56.79  132.25    4.53  24.25  174.26    32.22    25.56  27.29  58.64  28.23    57.85
```

```
2010-01-07   30.08   56.27  130.00   4.50  24.56  177.67   32.77   25.39  26.96  58.65  27.75   57.41
2010-01-08   30.28   56.77  133.52   4.52  24.66  174.31   33.15   25.40  27.05  59.30  27.57   60.17
2010-01-11   30.02   57.02  130.31   4.50  24.89  171.56   33.82   24.98  27.00  58.78  26.97   62.82
2010-01-12   29.67   56.03  127.35   4.47  24.78  167.82   39.17   24.97  26.55  57.04  26.77   62.40
2010-01-13   30.09   56.53  129.11   4.43  24.98  169.07   40.51   25.62  26.70  58.48  27.05   62.07
2010-01-14   29.92   56.16  127.35   4.47  24.95  168.53   39.03   26.05  26.76  59.46  26.95   62.20
2010-01-15   29.42   56.25  127.14   4.40  24.88  165.21   39.02   26.26  27.44  58.41  26.39   61.93
2010-01-19   30.72   57.55  127.61   4.44  25.00  166.86   38.09   26.48  27.99  59.95  26.86   62.25
2010-01-20   30.25   57.20  125.78   4.41  24.52  167.79   37.69   26.34  27.88  57.28  25.93   61.16
2010-01-21   29.72   56.63  126.62   4.35  26.76  160.87   37.54   25.85  27.39  53.24  25.41   59.70
2010-01-22   28.25   56.60  121.43   4.29  26.58  154.12   36.50   25.18  27.14  51.70  25.04   58.75
2010-01-25   29.01   55.71  120.31   4.34  26.58  154.98   36.45   25.24  27.56  52.48  25.18   58.75
2010-01-26   29.42   56.58  119.48   4.33  26.78  150.88   36.28   24.86  27.98  51.12  25.23   58.64
2010-01-27   29.70   57.74  122.75   4.33  26.82  151.50   37.04   25.63  27.36  50.94  25.18   59.34
2010-01-28   28.47   58.08  126.03   4.27  26.52  153.29   36.30   24.94  27.43  49.20  24.30   58.96
2010-01-29   27.44   58.48  125.41   4.22  26.26  148.72   36.69   24.99  26.54  48.50  23.87   57.77

 Contents of market_cap:
                   AAPL          AMGN          AMZN         CPRT           EL            GS       ILMN
MA            PAA            RIO            TEF           UPS
Date
2010-01-04  160386.7278  42475.580670  63893.145750  2090.225938  8892.669154  68854.242342  4469.465
28476.143688  19531.934838  101342.466626  143829.332465  50575.708420
2010-01-05  160701.5202  42107.635585  64270.110538  2090.225938  8859.692631  70071.563705  4440.205
28398.521801  19748.956336  102916.051241  143728.576365  50662.638135
2010-01-06  158130.7156  41791.202811  63105.814231  2081.038131  8885.341038  69323.666920  4713.786
28343.077596  19741.722286  106063.220471  142217.234868  50288.840359
2010-01-07  157815.9232  41408.539922  62032.180340  2067.256422  8998.926841  70680.224387  4794.251
28154.567299  19502.998638  106081.307650  139799.088473  49906.349611
2010-01-08  158865.2312  41776.485008  63711.820915  2076.444228  9035.567423  69343.557792  4849.845
28165.656140  19568.105088  107256.974316  138892.283574  52305.609756
```

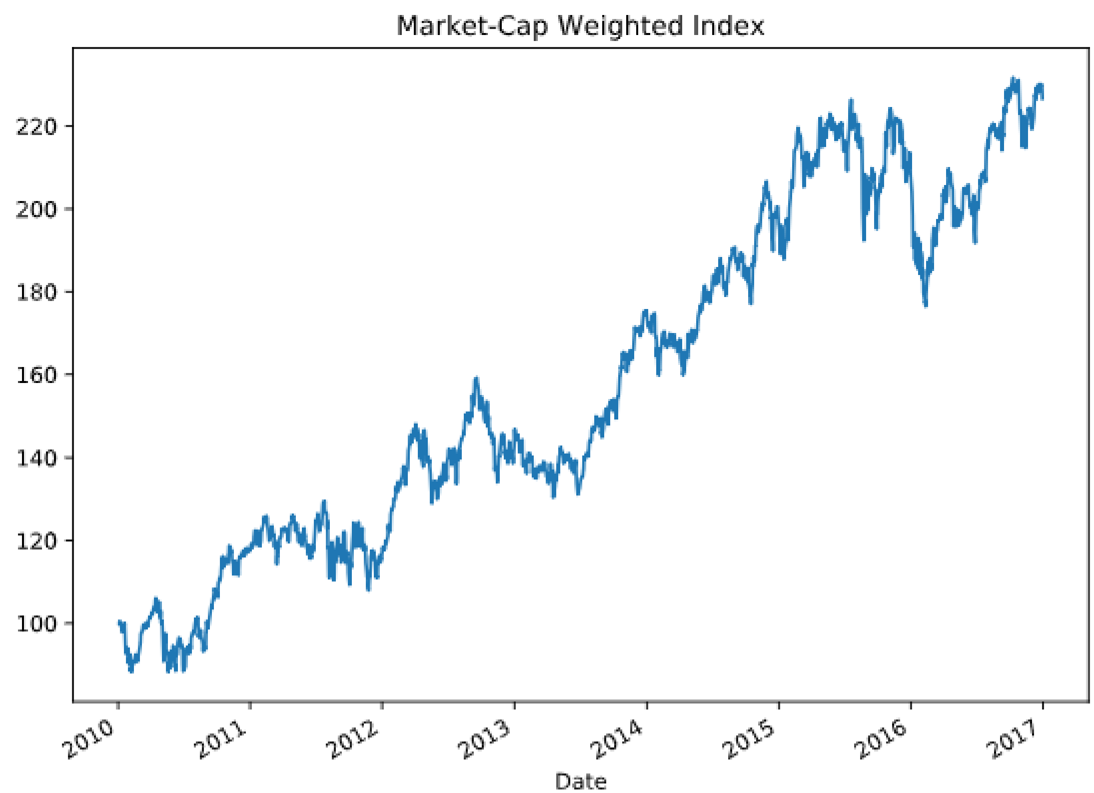


Calculate & plot the composite index

Use the time series of market capitalization created in the last exercise to aggregate the market value for each period, and then normalize this series to convert it to an index.

In [ ]:

```
# Aggregate and print the market cap per trading day
raw_index = market_cap_series.sum(axis=1)
# market_cap_series seems to be the market_cap in previous cell. It is a DataFrame for sure.

print(raw_index)

# Normalize the aggregate market cap here
index = raw_index.div(raw_index.iloc[0]).mul(100)
print(index)

# Plot the index here
index.plot(title='Market-Cap Weighted Index')
plt.show()
```



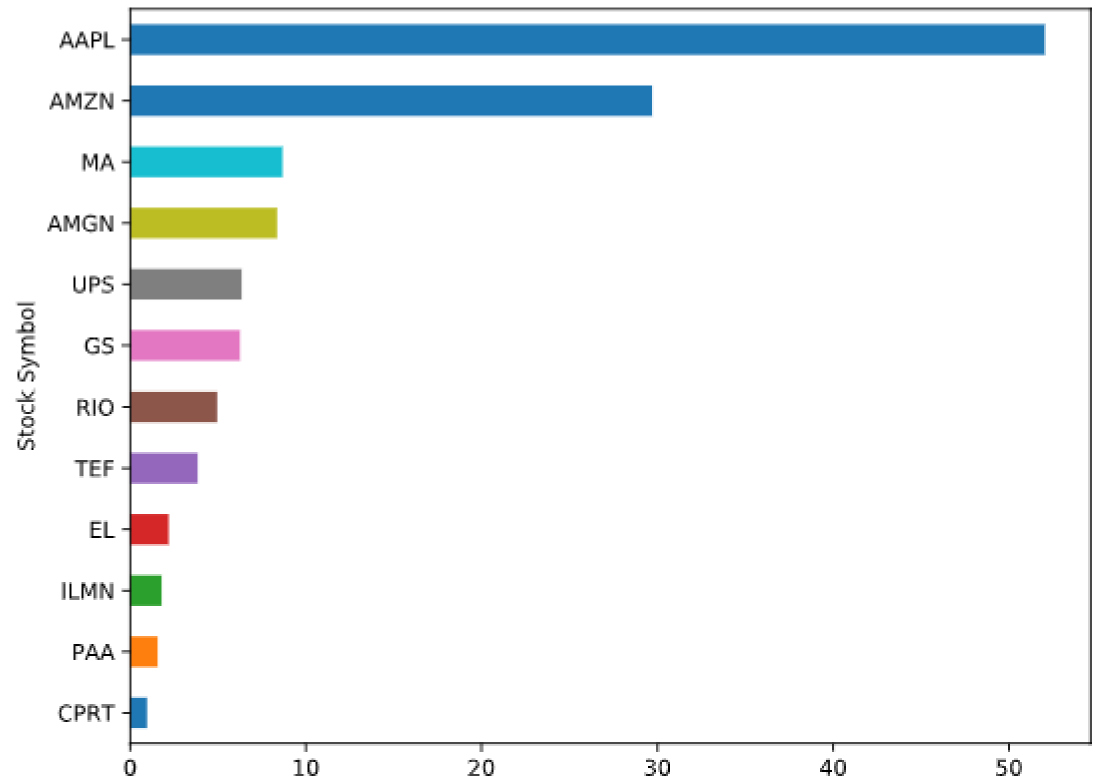
Calculate the contribution of each stock to the index

You have successfully built the value-weighted index. Let's now explore how it performed over the 2010-2016 period. Also determine how much each stock has contributed to the index return.

In [ ]:

```
# Calculate and print the index return here
index_return = (index.iloc[-1]/index.iloc[0] - 1) * 100
print(index_return)

# Select the market capitalization
market_cap = components['Market Capitalization']

# Calculate the total market cap
total_market_cap = market_cap.sum()

# Calculate the component weights, and print the result
weights = market_cap.div(total_market_cap)
print(weights.sort_values())

# Calculate and plot the contribution by component
weights.mul(index_return).sort_values().plot(kind='barh')
plt.show()
```

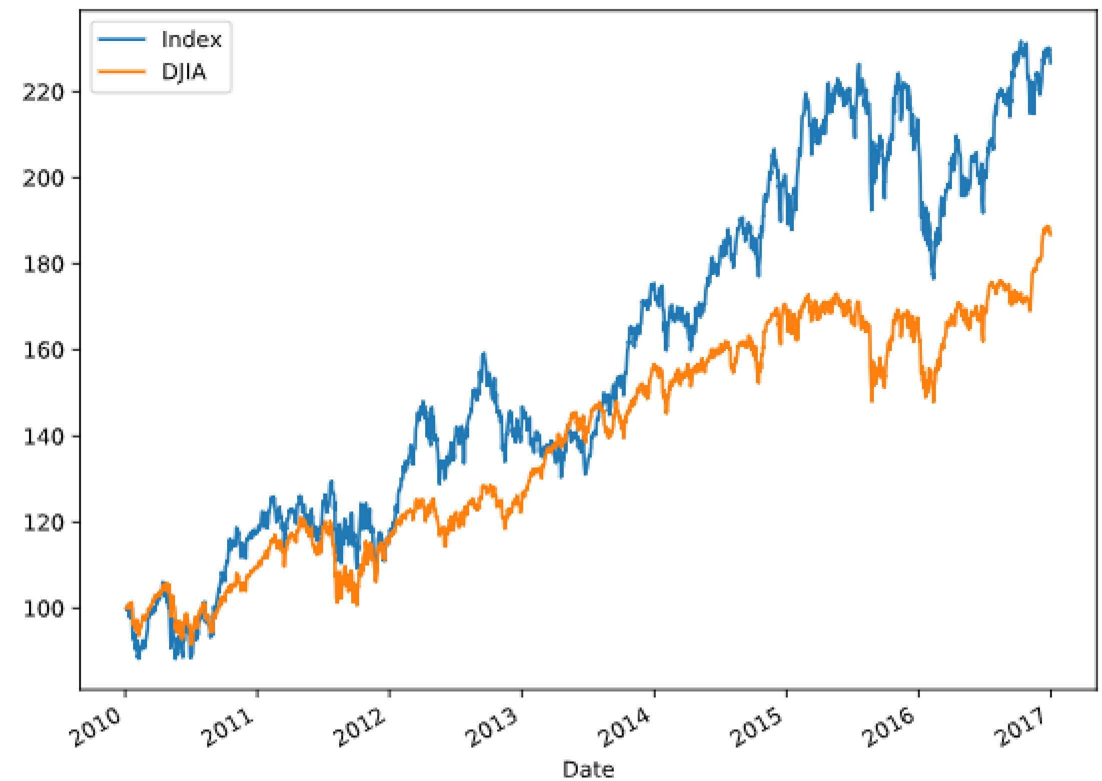
```
126.65826659999996
    Stock Symbol
    CPRT    0.007564
    PAA     0.012340
    ILMN    0.014110
    EL      0.017282
    TEF     0.030324
    RIO     0.039110
    GS      0.049332
    UPS     0.050077
    AMGN    0.066039
    MA      0.068484
    AMZN    0.234410
    AAPL    0.410929
    Name: Market Capitalization, dtype: float64
```



Compare index performance against benchmark I

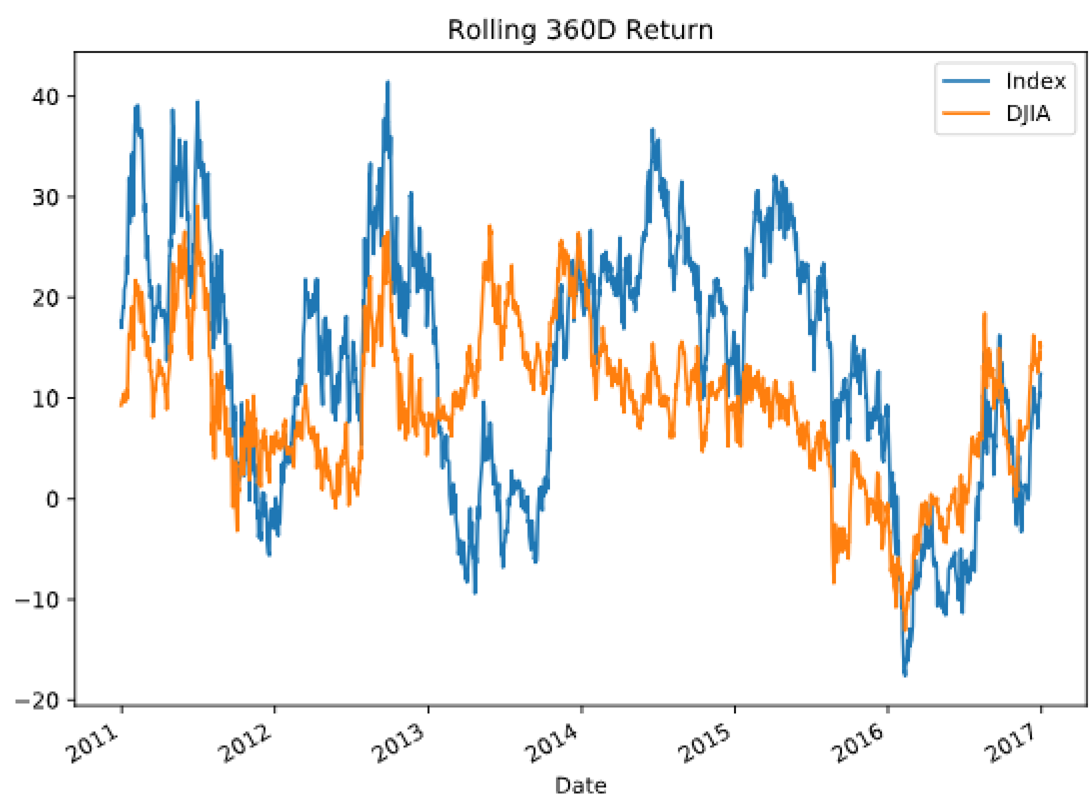
The next step in analyzing the performance of your index is to compare it against a benchmark.

In [ ]:
```
# Convert index series to dataframe here
data = index.to_frame('Index')

# Normalize djia series and add as new column to data
djia = djia.div(djia.iloc[0]).mul(100)
data['DJIA'] = djia

# Show total return for both index and djia
print(data.iloc[-1].div(data.iloc[0]).sub(1).mul(100))

# Plot both series
data.plot()
plt.show()
```



Compare index performance against benchmark II

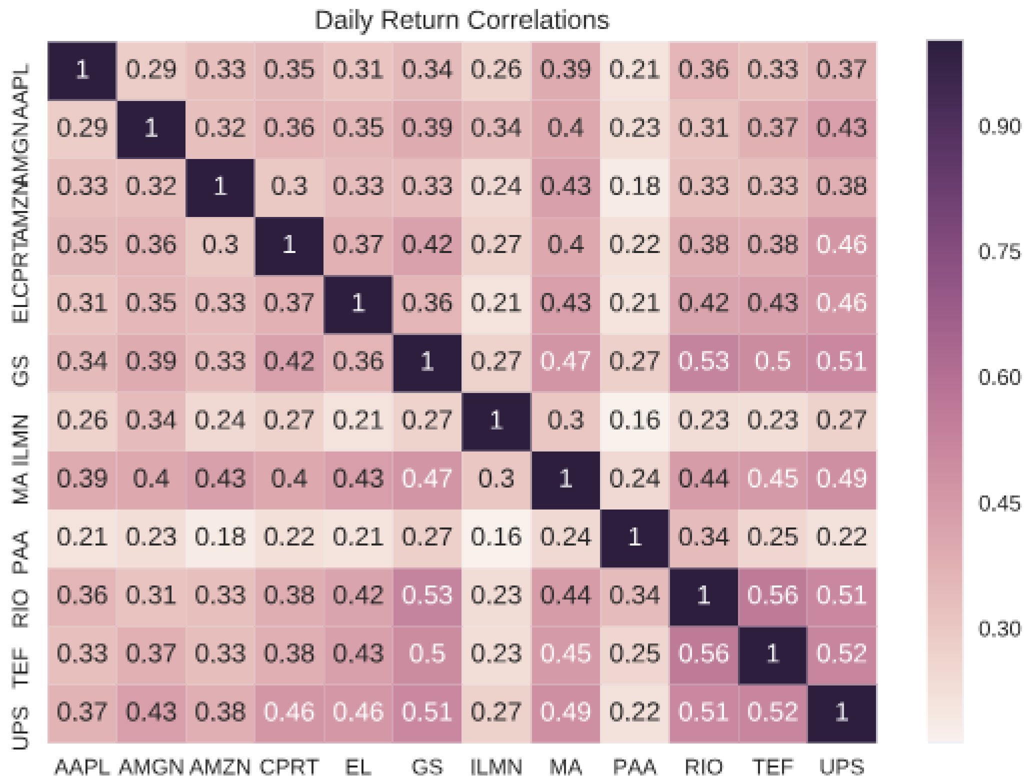
In [ ]:
```python
# Inspect data
print(data.info())
print(data.head())

# Create multi_period_return function here
def multi_period_return(r):
    return (np.prod(r + 1) - 1) * 100

# Calculate rolling_return_360
rolling_return_360 = data.pct_change().rolling('360D').apply(multi_period_return)

# Plot rolling_return_360 here
rolling_return_360.plot(title='Rolling 360D Return')
plt.show()
```



Visualize your index constituent correlations

To better understand the characteristics of your index constituents, you can calculate the return correlations.

In [ ]:
```
# Inspect stock_prices here
print(stock_prices.info())

# Calculate the daily returns
returns = stock_prices.pct_change()

# Calculate and print the pairwise correlations
correlations = returns.corr()
print(correlations)

# Plot a heatmap of daily return correlations
sns.heatmap(correlations, annot=True)
plt.title('Daily Return Correlations')
plt.show();
```



Save your analysis to multiple excel worksheets

In [ ]:

```
# Inspect index and stock_prices
print(index.info())
print(stock_prices.info())

# Join index to stock_prices, and inspect the result
data = stock_prices.join(index)
print(data.info())

# Create index & stock price returns
returns = data.pct_change()

# export data and data as returns to excel
with pd.ExcelWriter('data.xls') as writer:
    data.to_excel(writer, sheet_name='data')
    returns.to_excel(writer, sheet_name='returns')
```