How to use dates & times with pandas

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Date & time series functionality

- At the root: data types for date & time information
  - Objects for points in time and periods
  - Attributes & methods reflect time-related details
- Sequences of dates & periods:
  - Series or DataFrame columns
  - Index: convert object into Time Series
- Many Series/DataFrame methods rely on time information in the index to provide time-series functionality

Basic building block: pd.Timestamp

```
import pandas as pd  # assumed imported going forward
from datetime import datetime  # To manually create dates
time_stamp = pd.Timestamp(datetime(2017, 1, 1))
pd.Timestamp('2017-01-01') == time_stamp
```

```
True # Understands dates as strings
```

```
time_stamp # type: pandas.tslib.Timestamp
```

```
Timestamp('2017-01-01 00:00:00')
```

Basic building block: pd.Timestamp

- Timestamp object has many attributes to store time-specific information

```
time_stamp.year
```

```
2017
```

```
time_stamp.day_name()
```

```
'Sunday'
```

More building blocks: pd.Period & freq

```
period = pd.Period('2017-01')
period # default: month-end
```

```
Period('2017-01', 'M')
```

```
period.asfreq('D') # convert to daily
```

```
Period('2017-01-31', 'D')
```

```
period.to_timestamp().to_period('M')
```

```
Period('2017-01', 'M')
```

- Period object has freq attribute to store frequency info

- Convert `pd.Period()` to `pd.Timestamp()` and back

More building blocks: pd.Period & freq

```
period + 2
```

```
Period('2017-03', 'M')
```

```
pd.Timestamp('2017-01-31', 'M') + 1
```

```
Timestamp('2017-02-28 00:00:00', freq='M')
```

- Frequency info enables basic date arithmetic

Sequences of dates & times

- `pd.date_range` : `start` , `end` , `periods` , `freq`

```
index = pd.date_range(start='2017-1-1', periods=12, freq='M')
```

```
index
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31', ...,
               '2017-09-30', '2017-10-31', '2017-11-30', '2017-12-31'],
              dtype='datetime64[ns]', freq='M')
```

- `pd.DateTimeIndex` : sequence of Timestamp objects with frequency info

Sequences of dates & times

```
index[0]
```

```
Timestamp('2017-01-31 00:00:00', freq='M')
```

```
index.to_period()
```

```
PeriodIndex(['2017-01', '2017-02', '2017-03', '2017-04', ...,
             '2017-11', '2017-12'], dtype='period[M]', freq='M')
```

Create a time series: pd.DateTimeIndex

```
pd.DataFrame({'data': index}).info()
```

```
RangeIndex: 12 entries, 0 to 11
Data columns (total 1 columns):
data    12 non-null datetime64[ns]
dtypes: datetime64[ns](1)
```

Create a time series: pd.DateTimeIndex

- `np.random.random` :
  - Random numbers: `[0,1]`
  - 12 rows, 2 columns

```
data = np.random.random((size=12,2))
pd.DataFrame(data=data, index=index).info()
```

```
DatetimeIndex: 12 entries, 2017-01-31 to 2017-12-31
Freq: M
Data columns (total 2 columns):
0    12 non-null float64
1    12 non-null float64
dtypes: float64(2)
```

Frequency aliases & time info

There are many frequency aliases besides ‘M’ and ‘D’:

Period	Alias
Hour	H
Day	D
Week	W
Month	M
Quarter	Q
Year	A

These may be further differentiated by beginning/end of period, or business-specific definition

You can also access these **pd.Timestamp()** attributes:

attribute
.second, .minute, .hour,
.day, .month, .quarter, .year
.weekday
dayofweek
.weekofyear
.dayofyear

Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Indexing & resampling time series

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Time series transformation

Basic time series transformations include:

- Parsing string dates and convert to `datetime64`
- Selecting & slicing for specific subperiods
- Setting & changing `DateTimeIndex` frequency
  - Upsampling vs Downsampling

Getting GOOG stock prices

```
google = pd.read_csv('google.csv')  # import pandas as pd
google.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 504 entries, 0 to 503
Data columns (total 2 columns):
date     504 non-null object
price    504 non-null float64
dtypes: float64(1), object(1)
```

```
google.head()
```

```
         date   price
0  2015-01-02  524.81
1  2015-01-05  513.87
2  2015-01-06  501.96
3  2015-01-07  501.10
4  2015-01-08  502.68
```

Converting string dates to datetime64

- `pd.to_datetime()` :
  - Parse date string
  - Convert to `datetime64`

```
google.date = pd.to_datetime(google.date)
google.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 504 entries, 0 to 503
Data columns (total 2 columns):
date     504 non-null datetime64[ns]
price    504 non-null float64
dtypes: datetime64[ns](1), float64(1)
```

Converting string dates to datetime64

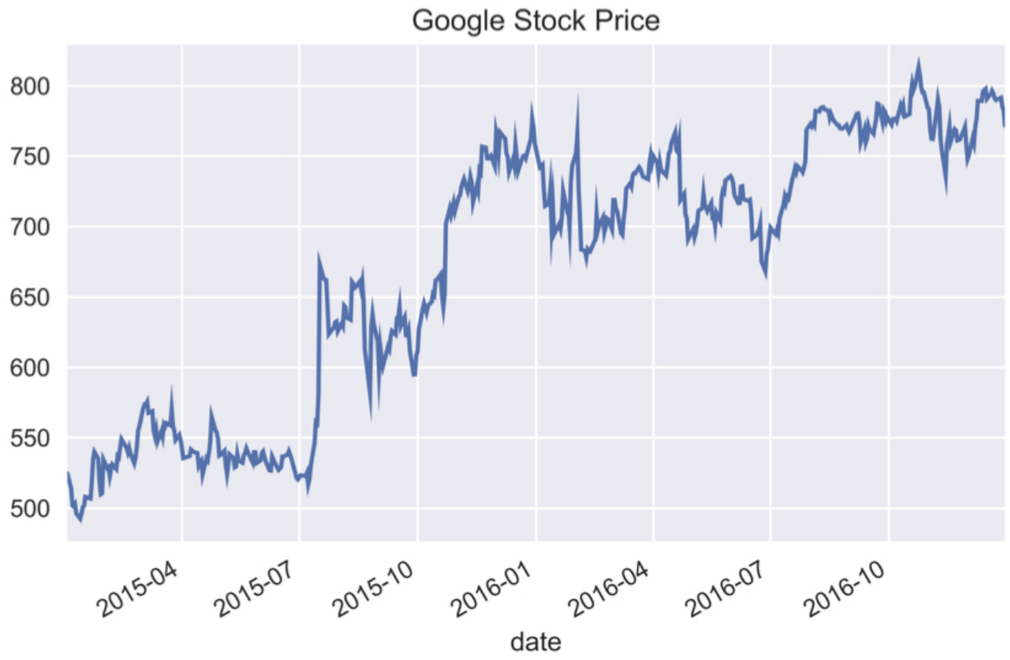
- `.set_index()` :
  - Date into index
  - `inplace` :
    - don't create copy

```
google.set_index('date', inplace=True)
google.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 504 entries, 2015-01-02 to 2016-12-30
Data columns (total 1 columns):
price    504 non-null float64
dtypes: float64(1)
```

Plotting the Google stock time series

```
google.price.plot(title='Google Stock Price')
plt.tight_layout(); plt.show()
```



Partial string indexing

- Selecting/indexing using strings that parse to dates

```
google['2015'].info() # Pass string for part of date
```

```
DatetimeIndex: 252 entries, 2015-01-02 to 2015-12-31
Data columns (total 1 columns):
price    252 non-null float64
dtypes: float64(1)
```

```
google['2015-3': '2016-2'].info() # Slice includes last month
```

```
DatetimeIndex: 252 entries, 2015-03-02 to 2016-02-29
Data columns (total 1 columns):
price    252 non-null float64
dtypes: float64(1)
memory usage: 3.9 KB
```

Partial string indexing

```
google.loc['2016-6-1', 'price'] # Use full date with .loc[]
```

```
734.15
```

.asfreq(): set frequency

- `.asfreq('D')`:
  - Convert `DateTimeIndex` to calendar day frequency

```
google.asfreq('D').info() # set calendar day frequency
```

```
DatetimeIndex: 729 entries, 2015-01-02 to 2016-12-30
Freq: D
Data columns (total 1 columns):
price    504 non-null float64
dtypes: float64(1)
```

.asfreq(): set frequency

- Upsampling:
  - Higher frequency implies new dates => missing data

```
google.asfreq('D').head()
```

```
              price
date
2015-01-02  524.81
2015-01-03     NaN
2015-01-04     NaN
2015-01-05  513.87
2015-01-06  501.96
```

.asfreq(): reset frequency

- `.asfreq('B')`:
  - Convert `DateTimeIndex` to business day frequency

```
google = google.asfreq('B') # Change to calendar day frequency
google.info()
```

```
DatetimeIndex: 521 entries, 2015-01-02 to 2016-12-30
Freq: B
Data columns (total 1 columns):
price    504 non-null float64
dtypes: float64(1)
```

.asfreq(): reset frequency

```
google[google.price.isnull()] # Select missing 'price' values
```

```
            price
date
2015-01-19    NaN
2015-02-16    NaN
...
2016-11-24    NaN
2016-12-26    NaN
```

- Business days that were not trading days

Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Lags, changes, and returns for stock price series

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at
Applied Artificial Intelligence

datacamp

Basic time series calculations

- Typical Time Series manipulations include:
  - Shift or lag values back or forward back in time
  - Get the difference in value for a given time period
  - Compute the percent change over any number of periods
- `pandas` built-in methods rely on `pd.DateTimeIndex`

Getting GOOG stock prices

- Let `pd.read_csv()` do the parsing for you!

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

```
google.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 504 entries, 2015-01-02 to 2016-12-30
Data columns (total 1 columns):
price    504 non-null float64
dtypes: float64(1)
```

Getting GOOG stock prices

```
google.head()
```

```
             price
date
2015-01-02  524.81
2015-01-05  513.87
2015-01-06  501.96
2015-01-07  501.10
2015-01-08  502.68
```

.shift(): Moving data between past & future

- `.shift()`:
  - defaults to `periods=1`
  - 1 period into future

```
google['shifted'] = google.price.shift() # default: periods=1
google.head(3)
```

```
             price  shifted
date
2015-01-02  542.81      NaN
2015-01-05  513.87   542.81
2015-01-06  501.96   513.87
```

.shift(): Moving data between past & future

- `.shift(periods=-1)` :
  - lagged data
  - 1 period back in time

```
google['lagged'] = google.price.shift(periods=-1)
google[['price', 'lagged', 'shifted']].tail(3)
```

```
             price  lagged  shifted
date
2016-12-28  785.05  782.79   791.55
2016-12-29  782.79  771.82   785.05
2016-12-30  771.82     NaN   782.79
```

Calculate one-period percent change

- x_t / x_{t-1}

```
google['change'] = google.price.div(google.shifted)
google[['price', 'shifted', 'change']].head(3)
```

```
             price  shifted    change
Date
2017-01-03  786.14      NaN       NaN
2017-01-04  786.90   786.14  1.000967
2017-01-05  794.02   786.90  1.009048
```

Calculate one-period percent change

```
google['return'] = google.change.sub(1).mul(100)
google[['price', 'shifted', 'change', 'return']].head(3)
```

```
            price  shifted  change  return
date
2015-01-02  524.81      NaN     NaN     NaN
2015-01-05  513.87   524.81    0.98   -2.08
2015-01-06  501.96   513.87    0.98   -2.32
```

.diff(): built-in time-series change

- Difference in value for two adjacent periods
- $x_t - x_{t-1}$

```
google['diff'] = google.price.diff()
google[['price', 'diff']].head(3)
```

```
             price        diff
date
2015-01-02  524.81         NaN
2015-01-05  513.87      -10.94
2015-01-06  501.96      -11.91
```

.pct_change(): built-in time-series % change

- Percent change for two adjacent periods
- $\frac{x_t}{x_{t-1}}$

```
google['pct_change'] = google.price.pct_change().mul(100)
google[['price', 'return', 'pct_change']].head(3)
```

```
              price    return  pct_change
date
2015-01-02  524.81       NaN         NaN
2015-01-05  513.87     -2.08       -2.08
2015-01-06  501.96     -2.32       -2.32
```

Looking ahead: Get multi-period returns

```
google['return_3d'] = google.price.pct_change(periods=3).mul(100)
google[['price', 'return_3d']].head()
```

```
             price  return_3d
date
2015-01-02  524.81         NaN
2015-01-05  513.87         NaN
2015-01-06  501.96         NaN
2015-01-07  501.10   -4.517825
2015-01-08  502.68   -2.177594
```

- Percent change for two periods, 3 trading days apart

Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Compare time series growth rates

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

Comparing stock performance

- Stock price series: hard to compare at different levels
- Simple solution: normalize price series to start at 100
- Divide all prices by first in series, multiply by 100
  - Same starting point
  - All prices relative to starting point
  - Difference to starting point in percentage points

Normalizing a single series (1)

```
google = pd.read_csv('google.csv', parse_dates=['date'], index_col='date')
google.head(3)
```

```
            price
date
2010-01-04  313.06
2010-01-05  311.68
2010-01-06  303.83
```

```
first_price = google.price.iloc[0]  # int-based selection
first_price
```

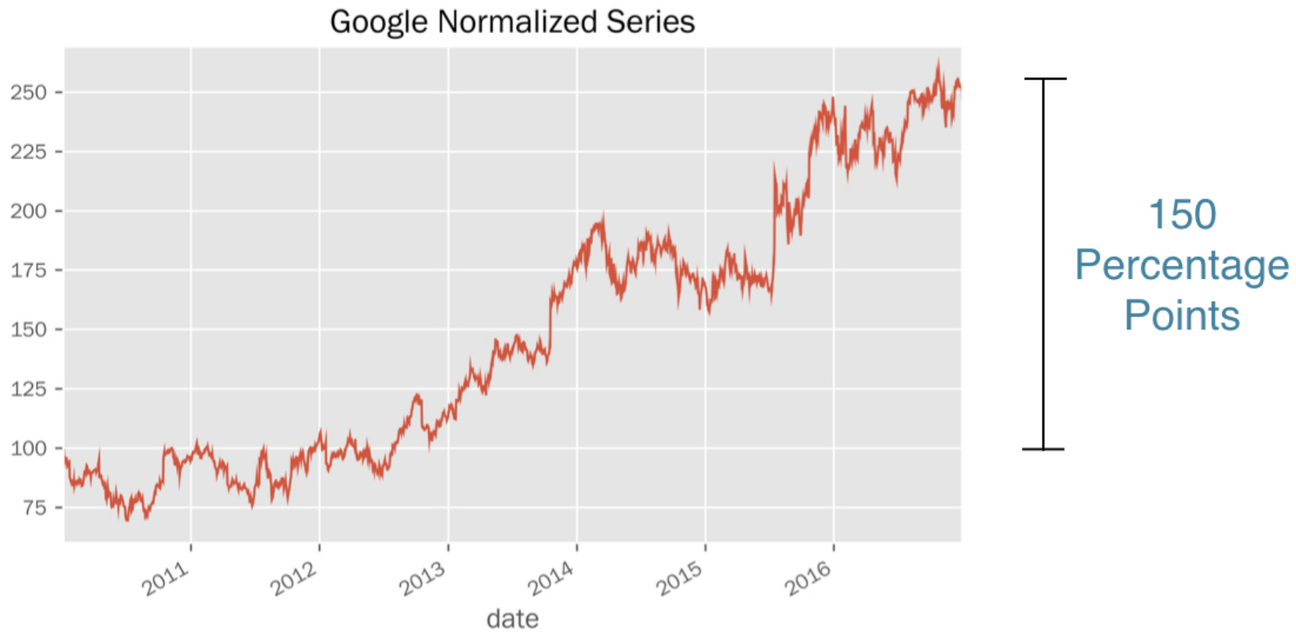
```
313.06
```

```
first_price == google.loc['2010-01-04', 'price']
```

```
True
```

Normalizing a single series (2)

```
normalized = google.price.div(first_price).mul(100)
normalized.plot(title='Google Normalized Series')
```



Normalizing multiple series (1)

```
prices = pd.read_csv('stock_prices.csv',
                     parse_dates=['date'],
                     index_col='date')
prices.info()
```

```
DatetimeIndex: 1761 entries, 2010-01-04 to 2016-12-30
Data columns (total 3 columns):
AAPL    1761 non-null float64
GOOG    1761 non-null float64
YHOO    1761 non-null float64
dtypes: float64(3)
```

```
prices.head(2)
```

```
             AAPL    GOOG   YHOO
Date
2010-01-04  30.57  313.06  17.10
2010-01-05  30.63  311.68  17.23
```

Normalizing multiple series (2)

```
prices.iloc[0]
```

```
AAPL     30.57
GOOG    313.06
YHOO     17.10
Name: 2010-01-04 00:00:00, dtype: float64
```

```
normalized = prices.div(prices.iloc[0])
normalized.head(3)
```

```
                AAPL      GOOG      YHOO
Date
2010-01-04  1.000000  1.000000  1.000000
2010-01-05  1.001963  0.995592  1.007602
2010-01-06  0.985934  0.970517  1.004094
```

- `.div()` : automatic alignment of Series index & DataFrame columns

Comparing with a benchmark (1)

```
index = pd.read_csv('benchmark.csv', parse_dates=['date'], index_col='date')
index.info()
```

```
DatetimeIndex: 1826 entries, 2010-01-01 to 2016-12-30
Data columns (total 1 columns):
SP500    1762 non-null float64
dtypes: float64(1)
```

```
prices = pd.concat([prices, index], axis=1).dropna()
prices.info()
```

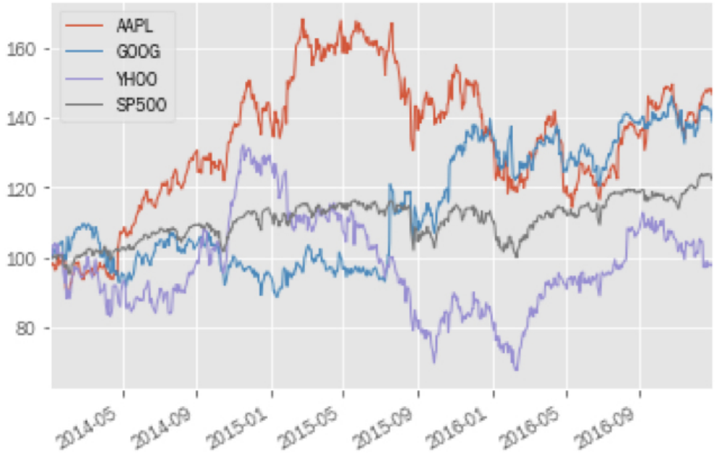
```
DatetimeIndex: 1761 entries, 2010-01-04 to 2016-12-30
Data columns (total 4 columns):
AAPL     1761 non-null float64
GOOG     1761 non-null float64
YHOO     1761 non-null float64
SP500    1761 non-null float64
dtypes: float64(4)
```

Comparing with a benchmark (2)

```
prices.head(1)
```

```
              AAPL    GOOG   YHOO     SP500
2010-01-04   30.57  313.06  17.10  1132.99
```

```
normalized = prices.div(prices.iloc[0]).mul(100)
normalized.plot()
```



Plotting performance difference

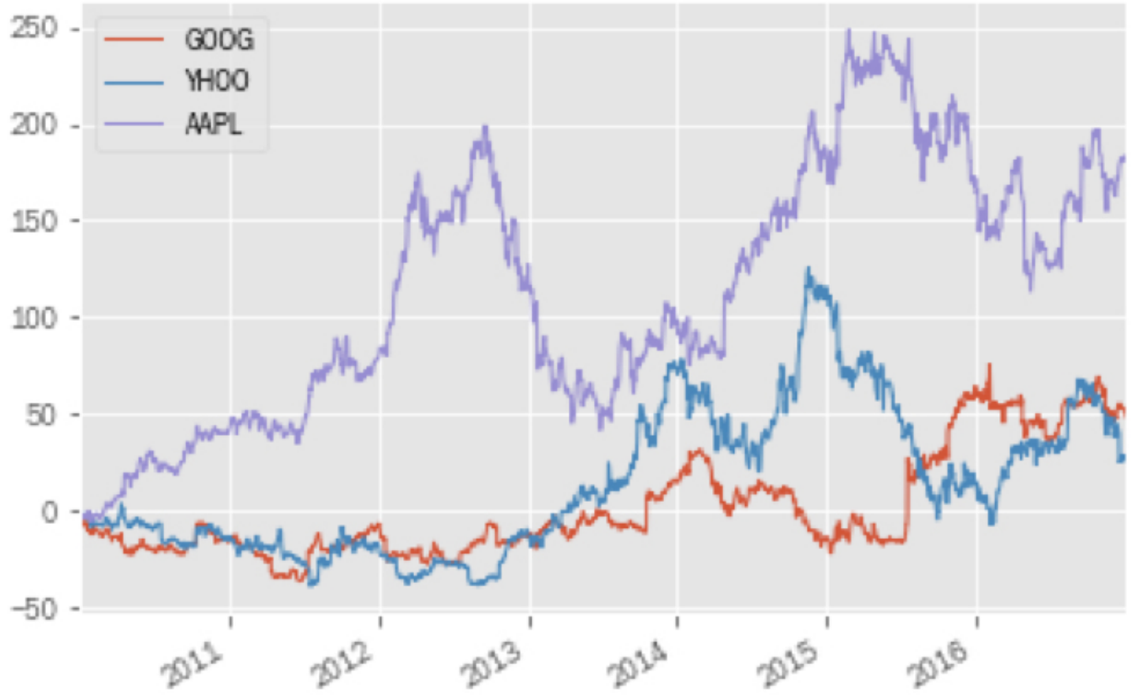
```
diff = normalized[tickers].sub(normalized['SP500'], axis=0)
```

```
                GOOG      YHOO      AAPL
2010-01-04  0.000000  0.000000  0.000000
2010-01-05 -0.752375  0.448669 -0.115294
2010-01-06 -3.314604  0.043069 -1.772895
```

- `.sub(..., axis=0)` : Subtract a Series from each DataFrame column by aligning indexes

Plotting performance difference

```
diff.plot()
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Changing the time series frequency: resampling

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Changing the frequency: resampling

- `DateTimeIndex` : set & change freq using `.asfreq()`
- But frequency conversion affects the data
  - Upsampling: fill or interpolate missing data
  - Downsampling: aggregate existing data
- `pandas` API:
  - `.asfreq()` , `.reindex()`
  - `.resample()` + transformation method

Getting started: quarterly data

```
dates = pd.date_range(start='2016', periods=4, freq='Q')
data = range(1, 5)
quarterly = pd.Series(data=data, index=dates)
quarterly
```

```
2016-03-31    1
2016-06-30    2
2016-09-30    3
2016-12-31    4
Freq: Q-DEC, dtype: int64 # Default: year-end quarters
```

Upsampling: quarter => month

```
monthly = quarterly.asfreq('M') # to month-end frequency
```

```
2016-03-31    1.0
2016-04-30    NaN
2016-05-31    NaN
2016-06-30    2.0
2016-07-31    NaN
2016-08-31    NaN
2016-09-30    3.0
2016-10-31    NaN
2016-11-30    NaN
2016-12-31    4.0
Freq: M, dtype: float64
```

- Upsampling creates missing values

```
monthly = monthly.to_frame('baseline') # to DataFrame
```

Upsampling: fill methods

```
monthly['ffill'] = quarterly.asfreq('M', method='ffill')
monthly['bfill'] = quarterly.asfreq('M', method='bfill')
monthly['value'] = quarterly.asfreq('M', fill_value=0)
```

Upsampling: fill methods

- `bfill` : backfill
- `ffill` : forward fill

	baseline	ffill	bfill	value
2016-03-31	1.0	1	1	1
2016-04-30	NaN	1	2	0
2016-05-31	NaN	1	2	0
2016-06-30	2.0	2	2	2
2016-07-31	NaN	2	3	0
2016-08-31	NaN	2	3	0
2016-09-30	3.0	3	3	3
2016-10-31	NaN	3	4	0
2016-11-30	NaN	3	4	0
2016-12-31	4.0	4	4	4

Add missing months: .reindex()

```
dates = pd.date_range(start='2016',
                      periods=12,
                      freq='M')
```

```
DatetimeIndex(['2016-01-31',
               '2016-02-29',
               ...,
               '2016-11-30',
               '2016-12-31'],
              dtype='datetime64[ns]', freq='M')
```

- `.reindex()` :
  - conform DataFrame to new index
  - same filling logic as `.asfreq()`

```
quarterly.reindex(dates)
```

```
2016-01-31    NaN
2016-02-29    NaN
2016-03-31    1.0
2016-04-30    NaN
2016-05-31    NaN
2016-06-30    2.0
2016-07-31    NaN
2016-08-31    NaN
2016-09-30    3.0
2016-10-31    NaN
2016-11-30    NaN
2016-12-31    4.0
```

Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Upsampling & interpolation with .resample()

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Frequency conversion & transformation methods

- `.resample()` : similar to `.groupby()`
- Groups data within resampling period and applies one or several methods to each group
- New date determined by offset - start, end, etc
- Upsampling: fill from existing or interpolate values
- Downsampling: apply aggregation to existing data

Getting started: monthly unemployment rate

```
unrate = pd.read_csv('unrate.csv', parse_dates['Date'], index_col='Date')
unrate.info()
```

```
DatetimeIndex: 208 entries, 2000-01-01 to 2017-04-01
Data columns (total 1 columns):
UNRATE    208 non-null float64 # no frequency information
dtypes: float64(1)
```

```
unrate.head()
```

```
            UNRATE
DATE
2000-01-01     4.0
2000-02-01     4.1
2000-03-01     4.0
2000-04-01     3.8
2000-05-01     4.0
```

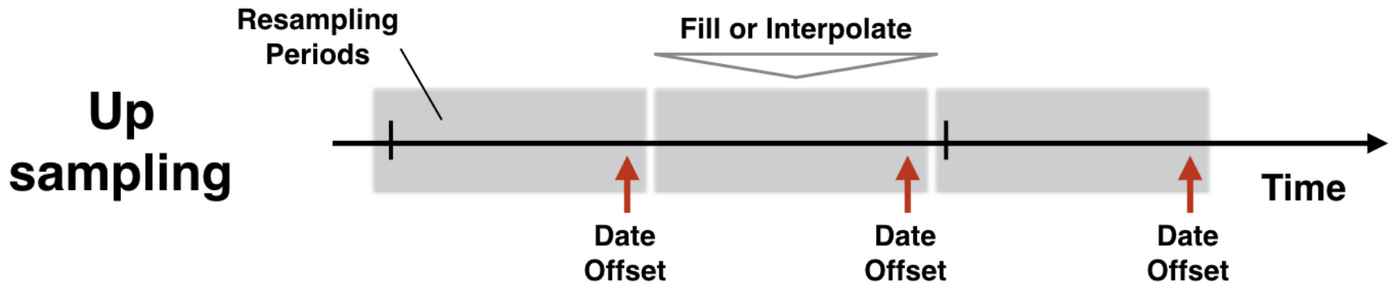
- Reporting date: 1st day of month

Resampling Period & Frequency Offsets

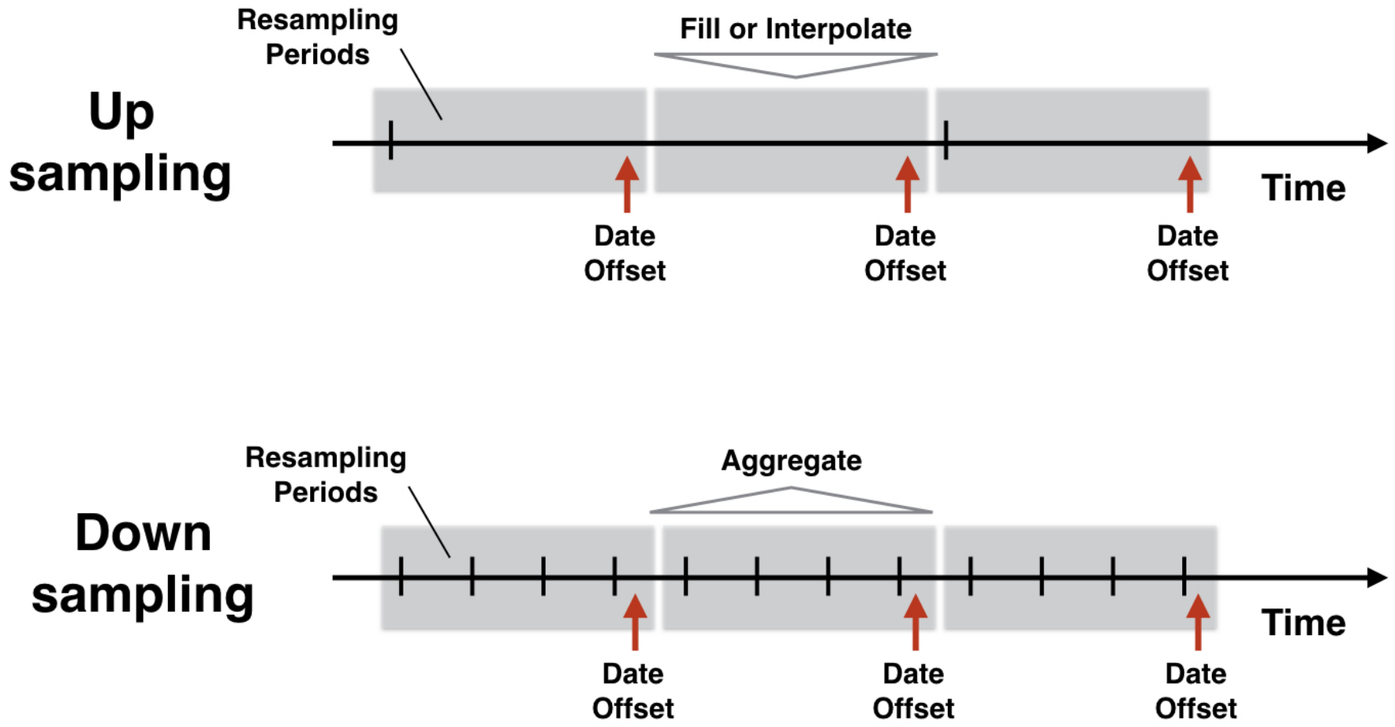
- Resample creates new date for frequency offset
- Several alternatives to calendar month end

Frequency	Alias	Sample Date
Calendar Month End	M	2017-04-30
Calendar Month Start	MS	2017-04-01
Business Month End	BM	2017-04-28
Business Month Start	BMS	2017-04-03

Resampling logic



Resampling logic



Assign frequency with .resample()

```
unrate.asfreq('MS').info()
```

```
DatetimeIndex: 208 entries, 2000-01-01 to 2017-04-01
Freq: MS
Data columns (total 1 columns):
UNRATE    208 non-null float64
dtypes: float64(1)
```

```
unrate.resample('MS') # creates Resampler object
```

```
DatetimeIndexResampler [freq=<MonthBegin>, axis=0, closed=left,
                        label=left, convention=start, base=0]
```

Assign frequency with .resample()

```
unrate.asfreq('MS').equals(unrate.resample('MS').asfreq())
```

```
True
```

- `.resample()` : returns data only when calling another method

Quarterly real GDP growth

```
gdp = pd.read_csv('gdp.csv')
gdp.info()
```

```
DatetimeIndex: 69 entries, 2000-01-01 to 2017-01-01
Data columns (total 1 columns):
gpd    69 non-null float64 # no frequency info
dtypes: float64(1)
```

```
gdp.head(2)
```

```
            gpd
DATE
2000-01-01  1.2
2000-04-01  7.8
```

Interpolate monthly real GDP growth

```
gdp_1 = gdp.resample('MS').ffill().add_suffix('_ffill')
```

```
            gpd_ffill
DATE
2000-01-01  1.2
2000-02-01  1.2
2000-03-01  1.2
2000-04-01  7.8
```

Interpolate monthly real GDP growth

```
gdp_2 = gdp.resample('MS').interpolate().add_suffix('_inter')
```

```
            gpd_inter
DATE
2000-01-01  1.200000
2000-02-01  3.400000
2000-03-01  5.600000
2000-04-01  7.800000
```

- `.interpolate()` : finds points on straight line between existing data

Concatenating two DataFrames

```
df1 = pd.DataFrame([1, 2, 3], columns=['df1'])
df2 = pd.DataFrame([4, 5, 6], columns=['df2'])
pd.concat([df1, df2])
```

```
   df1  df2
0  1.0  NaN
1  2.0  NaN
2  3.0  NaN
0  NaN  4.0
1  NaN  5.0
2  NaN  6.0
```

Concatenating two DataFrames

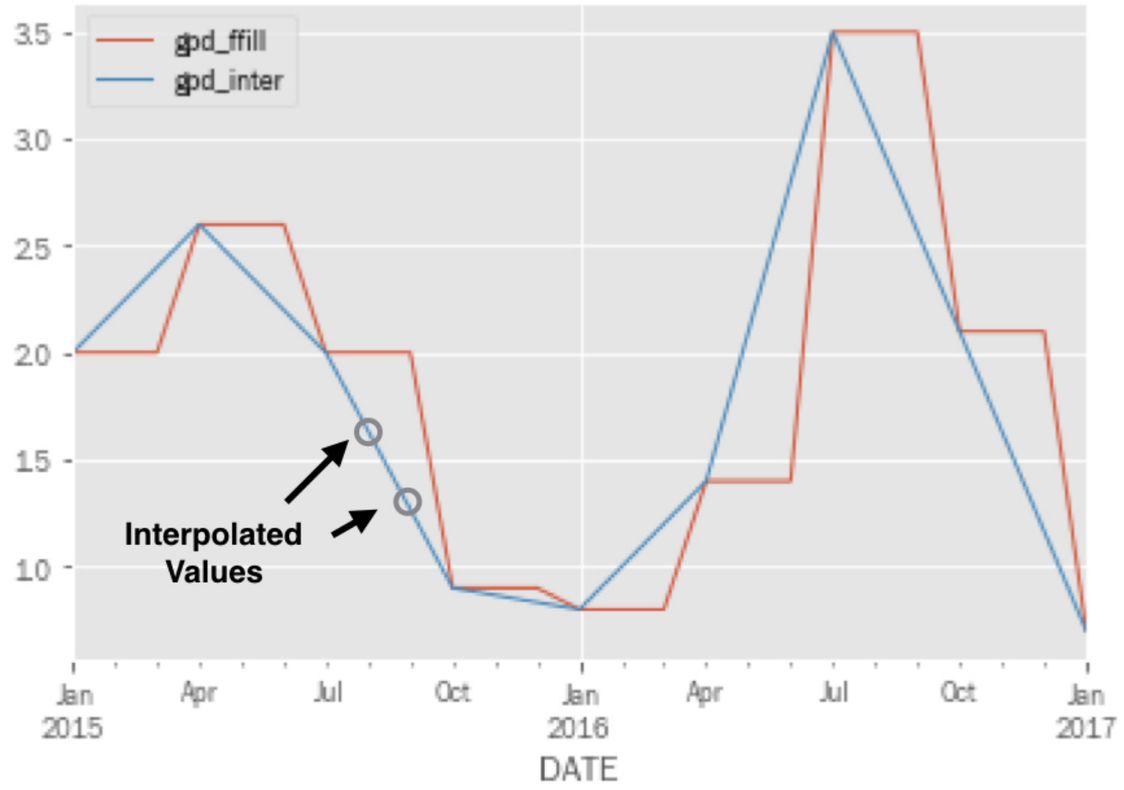
```
pd.concat([df1, df2], axis=1)
```

```
   df1  df2
0    1    4
1    2    5
2    3    6
```

- `axis=1` : concatenate horizontally

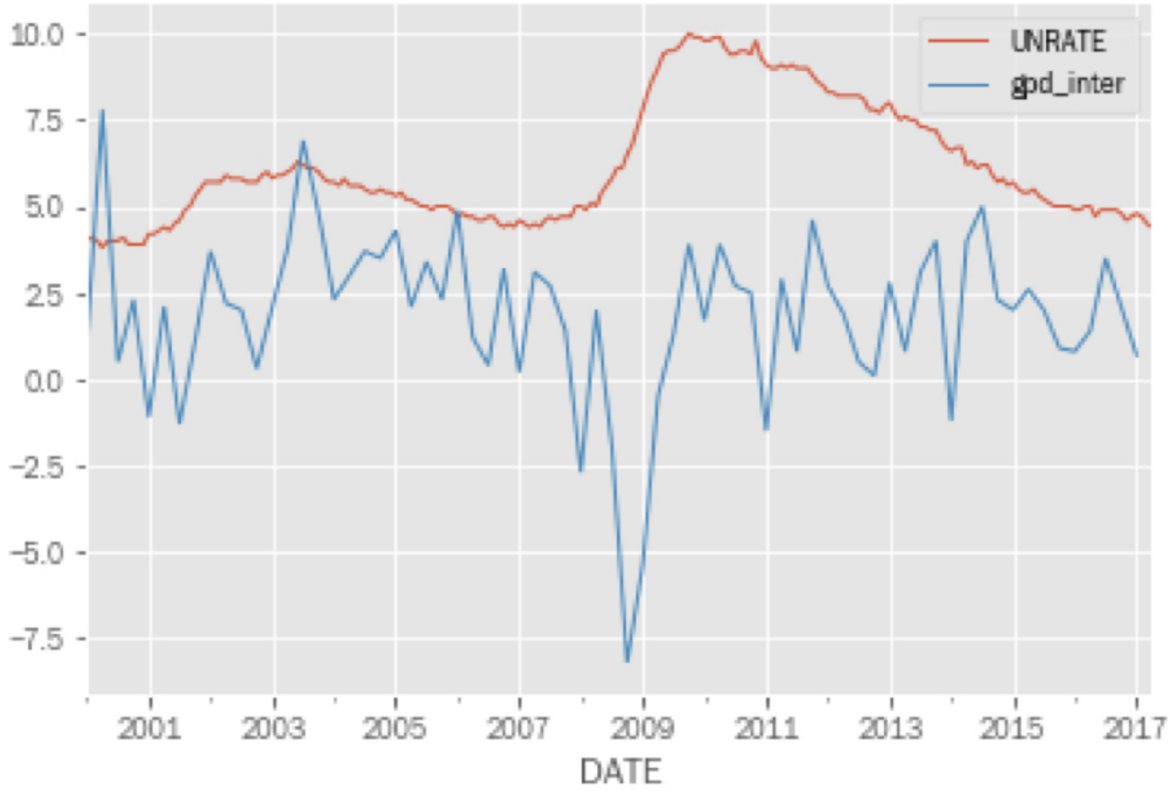
Plot interpolated real GDP growth

```
pd.concat([gdp_1, gdp_2], axis=1).loc['2015':].plot()
```



Combine GDP growth & unemployment

```
pd.concat([unrate, gdp_inter], axis=1).plot();
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Downsampling & aggregation

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

Downsampling & aggregation methods

- So far: upsampling, fill logic & interpolation
- Now: downsampling
  - hour to day
  - day to month, etc
- How to represent the existing values at the new date?
  - Mean, median, last value?

Air quality: daily ozone levels

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

```
DatetimeIndex: 6291 entries, 2000-01-01 to 2017-03-31
Data columns (total 1 columns):
Ozone    6167 non-null float64
dtypes: float64(1)
```

```
ozone = ozone.resample('D').asfreq()
ozone.info()
```

```
DatetimeIndex: 6300 entries, 1998-01-05 to 2017-03-31
Freq: D
Data columns (total 1 columns):
Ozone    6167 non-null float64
dtypes: float64(1)
```

Creating monthly ozone data

```
ozone.resample('M').mean().head()
```

```
            Ozone
date
2000-01-31  0.010443
2000-02-29  0.011817
2000-03-31  0.016810
2000-04-30  0.019413
2000-05-31  0.026535
```

```
ozone.resample('M').median().head()
```

```
            Ozone
date
2000-01-31  0.009486
2000-02-29  0.010726
2000-03-31  0.017004
2000-04-30  0.019866
2000-05-31  0.026018
```

`.resample().mean()` : Monthly average, assigned to end of calendar month

Creating monthly ozone data

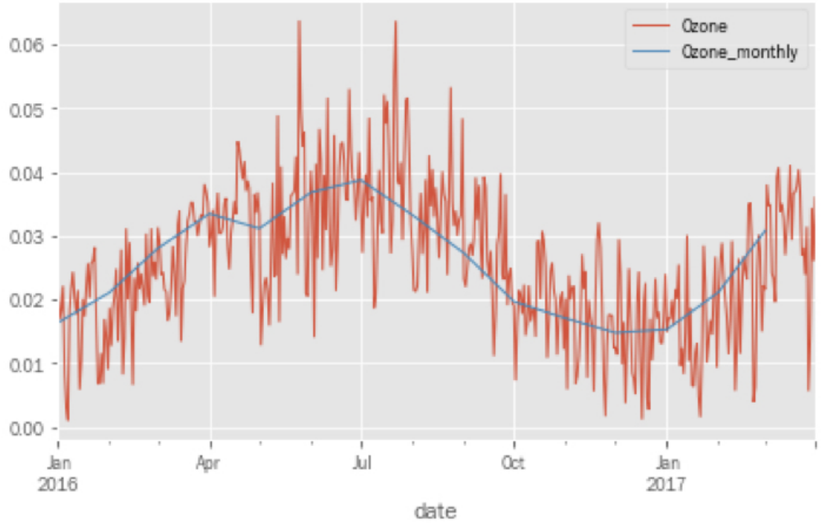
```
ozone.resample('M').agg(['mean', 'std']).head()
```

```
              Ozone
               mean       std
date
2000-01-31  0.010443  0.004755
2000-02-29  0.011817  0.004072
2000-03-31  0.016810  0.004977
2000-04-30  0.019413  0.006574
2000-05-31  0.026535  0.008409
```

- `.resample().agg()` : List of aggregation functions like groupby

Plotting resampled ozone data

```
ozone = ozone.loc['2016':]
ax = ozone.plot()
monthly = ozone.resample('M').mean()
monthly.add_suffix('_monthly').plot(ax=ax)
```



ax=ax:
Matplotlib let's you plot again on the axes object returned by the first plot

Resampling multiple time series

```
data = pd.read_csv('ozone_pm25.csv',
                   parse_dates=['date'],
                   index_col='date')
data = data.resample('D').asfreq()
data.info()
```

```
DatetimeIndex: 6300 entries, 2000-01-01 to 2017-03-31
Freq: D
Data columns (total 2 columns):
Ozone    6167 non-null float64
PM25     6167 non-null float64
dtypes: float64(2)
```

Resampling multiple time series

```
data = data.resample('BM').mean()
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 207 entries, 2000-01-31 to 2017-03-31
Freq: BM
Data columns (total 2 columns):
ozone    207 non-null float64
pm25     207 non-null float64
dtypes: float64(2)
```

Resampling multiple time series

```
df.resample('M').first().head(4)
```

```
              Ozone       PM25
date
2000-01-31  0.005545  20.800000
2000-02-29  0.016139   6.500000
2000-03-31  0.017004   8.493333
2000-04-30  0.031354   6.889474
```

```
df.resample('MS').first().head()
```

```
              Ozone       PM25
date
2000-01-01  0.004032  37.320000
2000-02-01  0.010583  24.800000
2000-03-01  0.007418  11.106667
2000-04-01  0.017631  11.700000
2000-05-01  0.022628   9.700000
```

Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Rolling window functions with pandas

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

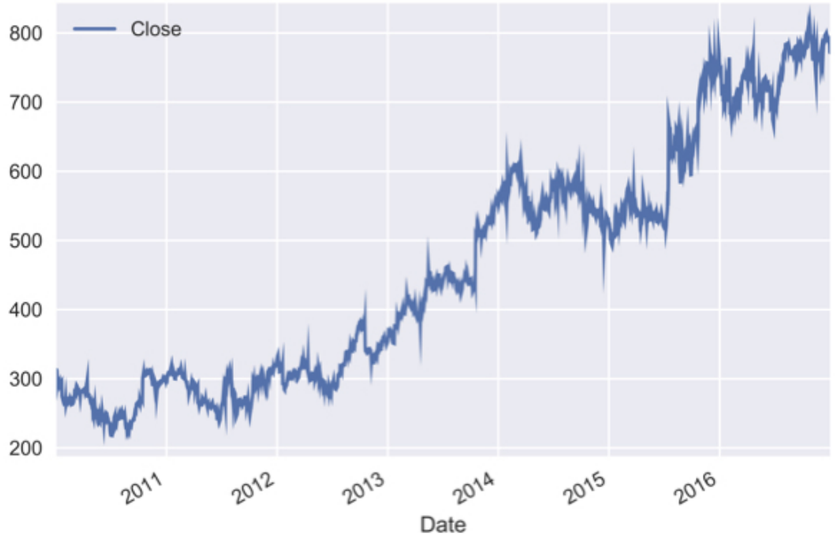
Window functions in pandas

- Windows identify sub periods of your time series
- Calculate metrics for sub periods inside the window
- Create a new time series of metrics
- Two types of windows:
  - Rolling: same size, sliding (this video)
  - Expanding: contain all prior values (next video)

Calculating a rolling average

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

```
DatetimeIndex: 1761 entries, 2010-01-04 to 2016-12-30
Data columns (total 1 columns):
price     1761 non-null float64
dtypes: float64(1)
```



Calculating a rolling average

```
# Integer-based window size
data.rolling(window=30).mean() # fixed # observations
```

```
DatetimeIndex: 1761 entries, 2010-01-04 to 2017-05-24
Data columns (total 1 columns):
price    1732 non-null float64
dtypes: float64(1)
```

- `window=30` : # business days
- `min_periods` : choose value < 30 to get results for first days

Calculating a rolling average

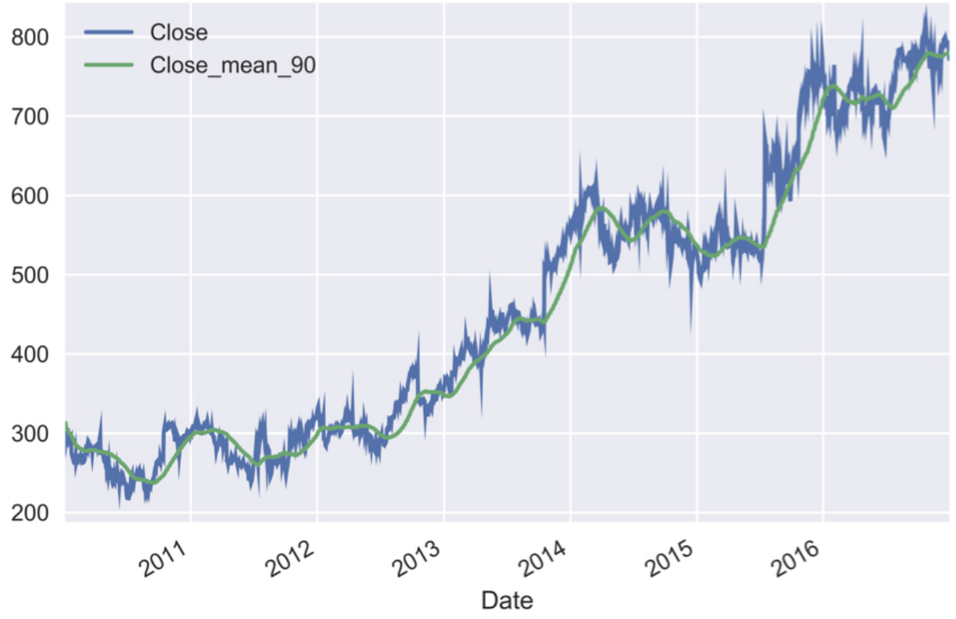
```
# Offset-based window size
data.rolling(window='30D').mean() # fixed period length
```

```
DatetimeIndex: 1761 entries, 2010-01-04 to 2017-05-24
Data columns (total 1 columns):
price    1761 non-null float64
dtypes: float64(1)
```

- `30D` : # calendar days

90 day rolling mean

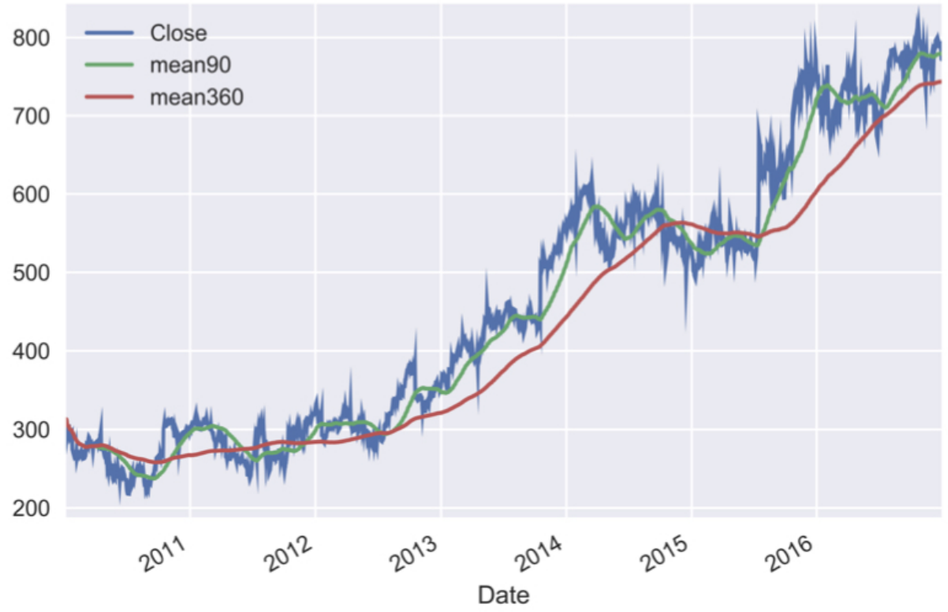
```
r90 = data.rolling(window='90D').mean()
google.join(r90.add_suffix('_mean_90')).plot()
```



.join:
concatenate Series **or** DataFrame **along** axis=1

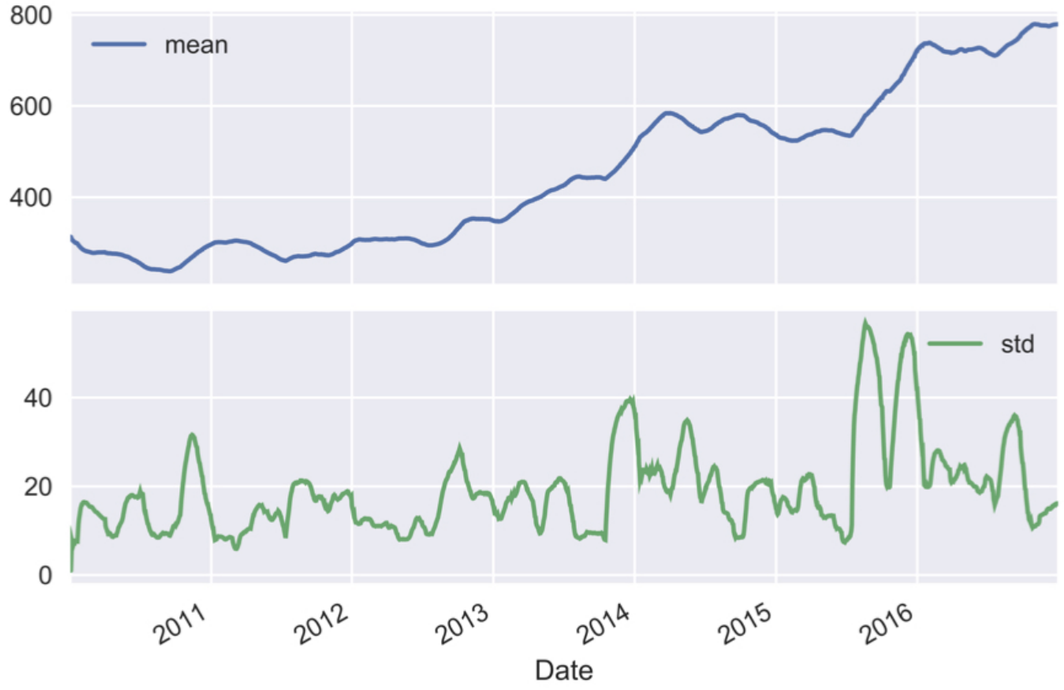
90 & 360 day rolling means

```
data['mean90'] = r90
r360 = data['price'].rolling(window='360D'.mean()
data['mean360'] = r360; data.plot()
```



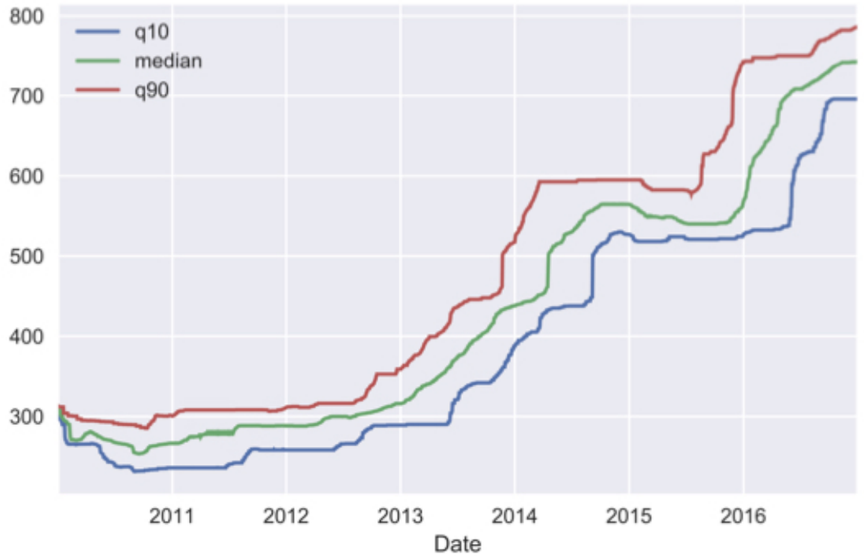
Multiple rolling metrics (1)

```
r = data.price.rolling('90D').agg(['mean', 'std'])
r.plot(subplots = True)
```



Multiple rolling metrics (2)

```
rolling = data.google.rolling('360D')
q10 = rolling.quantile(0.1).to_frame('q10')
median = rolling.median().to_frame('median')
q90 = rolling.quantile(0.9).to_frame('q90')
pd.concat([q10, median, q90], axis=1).plot()
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Expanding window functions with pandas

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Expanding windows in pandas

- From rolling to expanding windows
- Calculate metrics for periods up to current date
- New time series reflects all historical values
- Useful for running rate of return, running min/max
- Two options with pandas:
  - `.expanding()` - just like `.rolling()`
  - `.cumsum()`, `.cumprod()`, `cummin()` / `max()`

The basic idea

```
df = pd.DataFrame({'data': range(5)})
df['expanding sum'] = df.data.expanding().sum()
df['cumulative sum'] = df.data.cumsum()
df
```

```
   data  expanding sum  cumulative sum
0     0            0.0               0
1     1            1.0               1
2     2            3.0               3
3     3            6.0               6
4     4           10.0              10
```

Get data for the S&P 500

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

```
DatetimeIndex: 2519 entries, 2007-05-24 to 2017-05-24
Data columns (total 1 columns):
SP500    2519 non-null float64
```

How to calculate a running return

- Single period return r_t: current price over last price minus 1:

$$r_t = \frac{P_t}{P_{t-1}} - 1$$

  - Multi-period return: product of $(1 + r_t)$ for all periods, minus 1:

$$R_T = (1 + r_1)(1 + r_2)...(1 + r_T) - 1$$

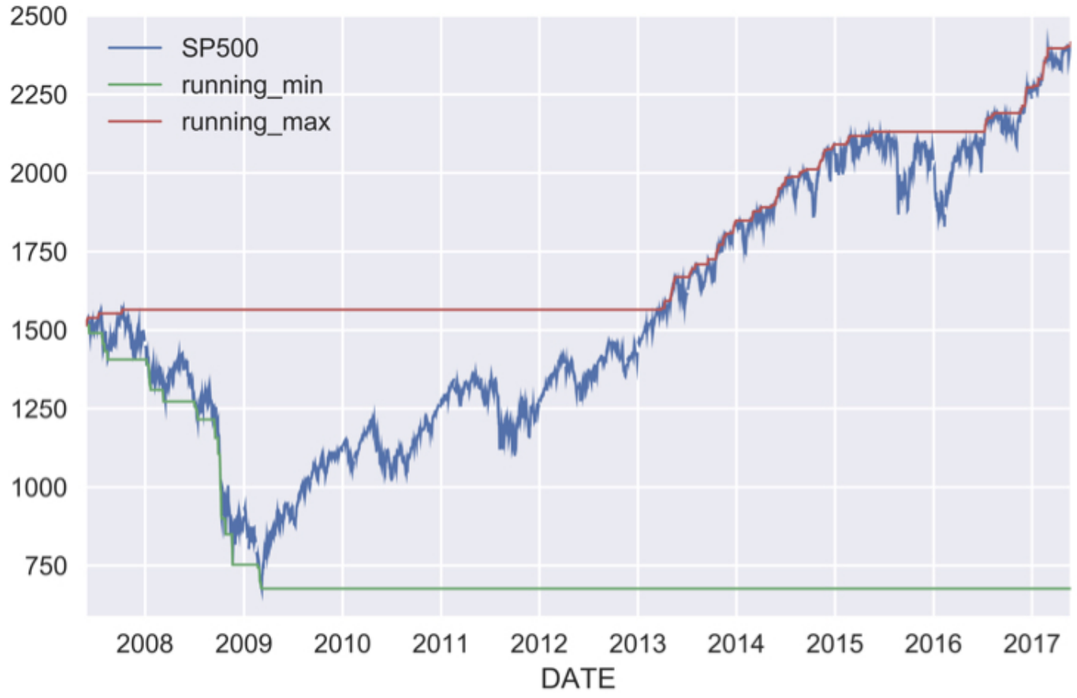
  - For the period return: `.pct_change()`
  - For basic math `.add()`, `.sub()`, `.mul()`, `.div()`
  - For cumulative product: `.cumprod()`

Running rate of return in practice

```
pr = data.SP500.pct_change() # period return
pr_plus_one = pr.add(1)
cumulative_return = pr_plus_one.cumprod().sub(1)
cumulative_return.mul(100).plot()
```

Getting the running min & max

```
data['running_min'] = data.SP500.expanding().min()
data['running_max'] = data.SP500.expanding().max()
data.plot()
```

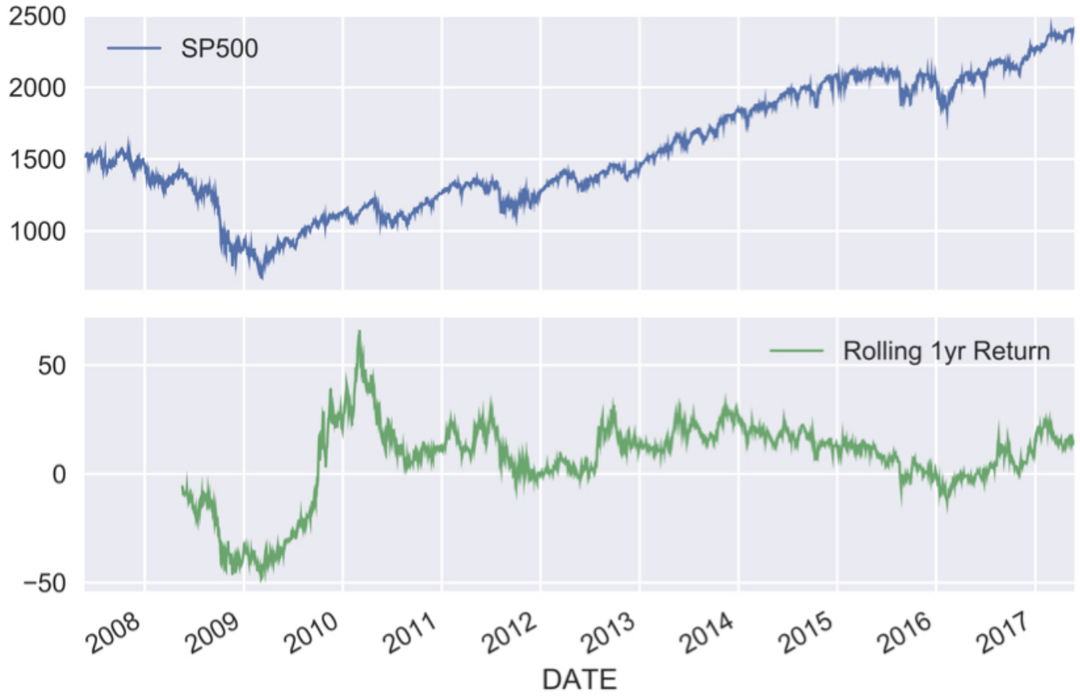


Rolling annual rate of return

```
def multi_period_return(period_returns):
    return np.prod(period_returns + 1) - 1
pr = data.SP500.pct_change() # period return
r = pr.rolling('360D').apply(multi_period_return)
data['Rolling 1yr Return'] = r.mul(100)
data.plot(subplots=True)
```

Rolling annual rate of return

```
data['Rolling 1yr Return'] = r.mul(100)
data.plot(subplots=True)
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Case study: S&P500 price simulation

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

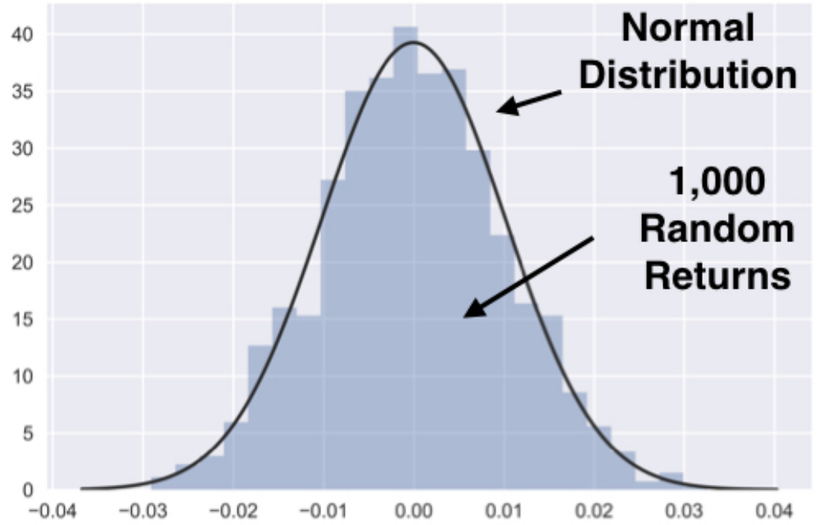
datacamp

Random walks & simulations

- Daily stock returns are hard to predict
- Models often assume they are random in nature
- Numpy allows you to generate random numbers
- From random returns to prices: use `.cumprod()`
- Two examples:
  - Generate random returns
  - Randomly selected actual SP500 returns

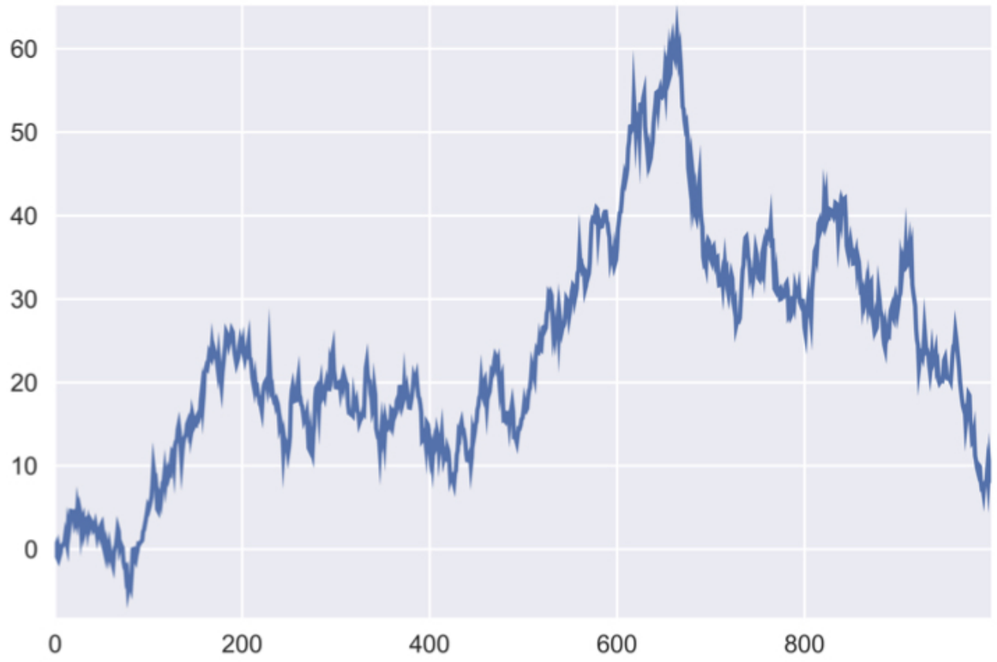
Generate random numbers

```
from numpy.random import normal, seed
from scipy.stats import norm
seed(42)
random_returns = normal(loc=0, scale=0.01, size=1000)
sns.distplot(random_returns, fit=norm, kde=False)
```



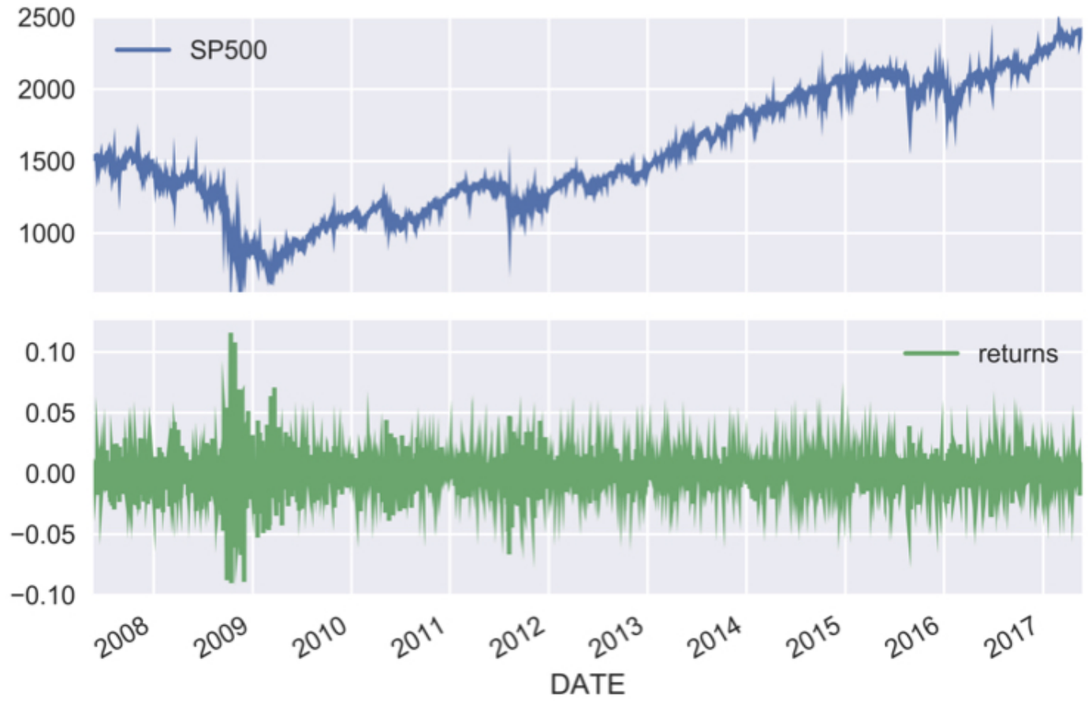
Create a random price path

```
return_series = pd.Series(random_returns)
random_prices = return_series.add(1).cumprod().sub(1)
random_prices.mul(100).plot()
```



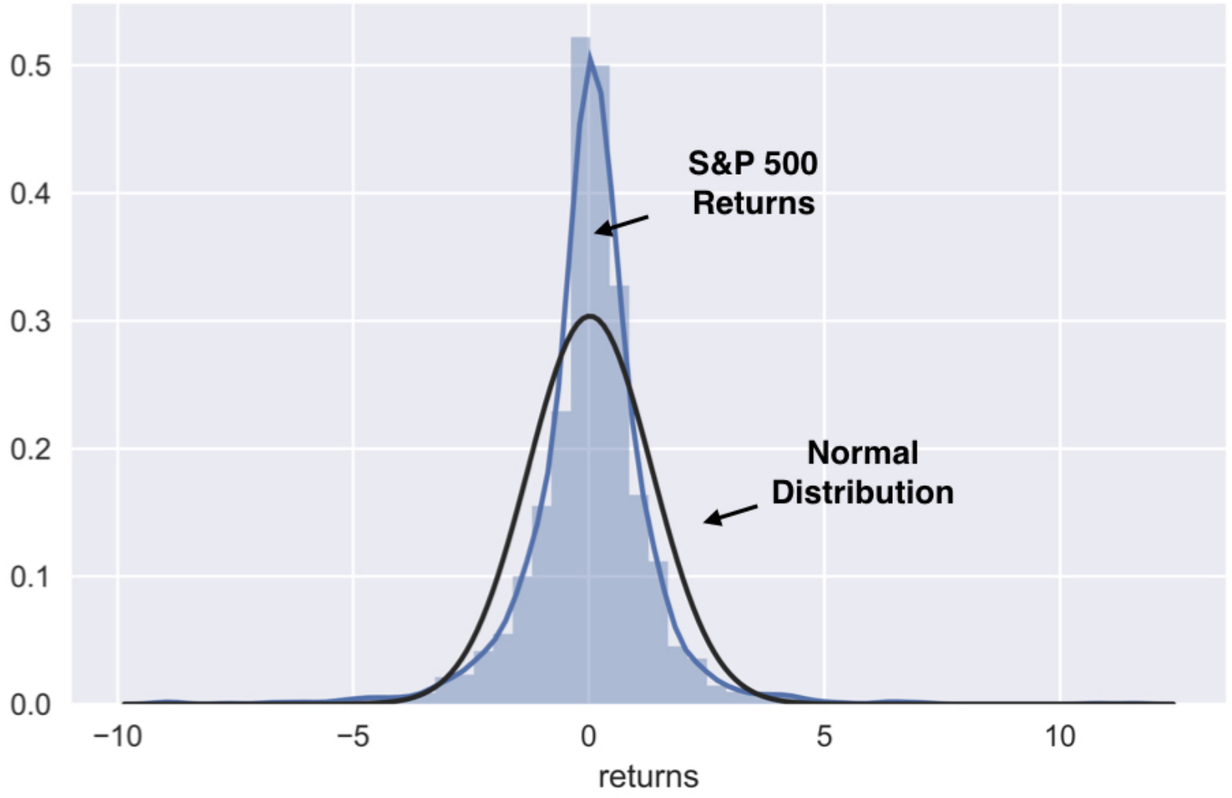
S&P 500 prices & returns

```
data = pd.read_csv('sp500.csv', parse_dates=['date'], index_col='date')
data['returns'] = data.SP500.pct_change()
data.plot(subplots=True)
```



S&P return distribution

```
sns.distplot(data.returns.dropna().mul(100), fit=norm)
```



Generate random S&P 500 returns

```
from numpy.random import choice
sample = data.returns.dropna()
n_obs = data.returns.count()
random_walk = choice(sample, size=n_obs)
random_walk = pd.Series(random_walk, index=sample.index)
random_walk.head()
```

```
DATE
2007-05-29   -0.008357
2007-05-30    0.003702
2007-05-31   -0.013990
2007-06-01    0.008096
2007-06-04    0.013120
```

Random S&P 500 prices (1)

```
start = data.SP500.first('D')
```

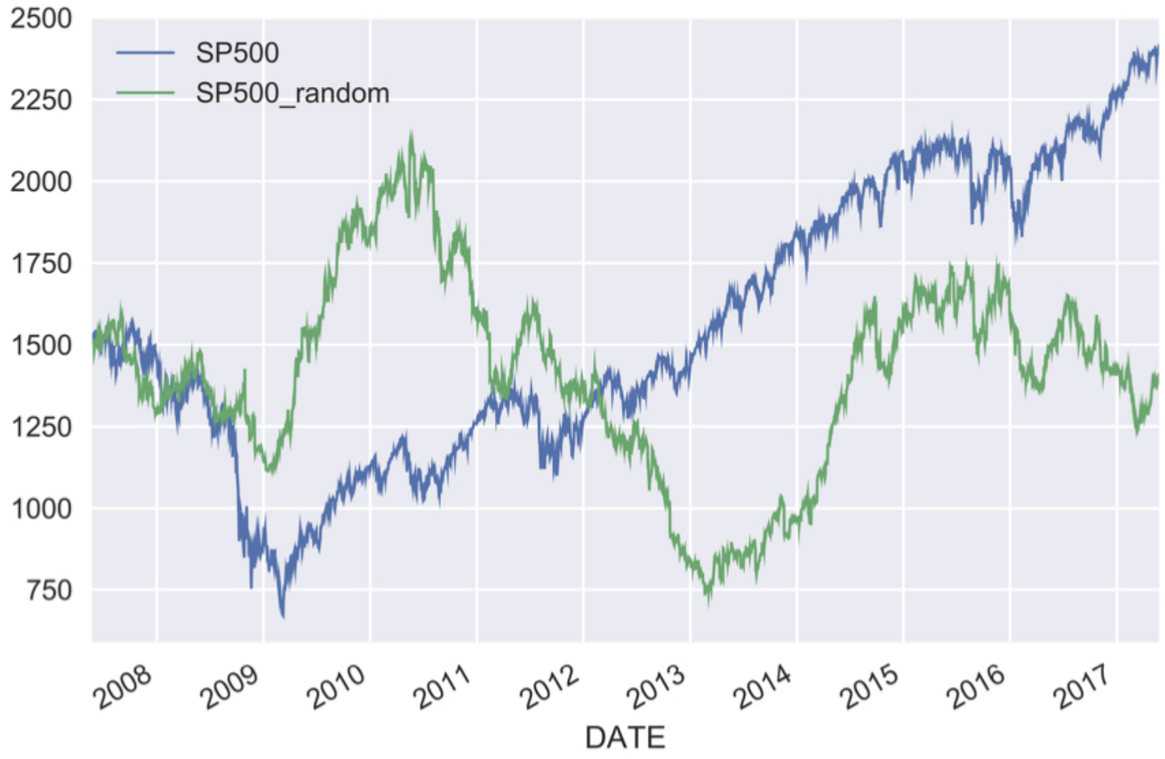
```
DATE
2007-05-25    1515.73
Name: SP500, dtype: float64
```

```
sp500_random = start.append(random_walk.add(1))
sp500_random.head())
```

```
DATE
2007-05-25    1515.730000
2007-05-29       0.998290
2007-05-30       0.995190
2007-05-31       0.997787
2007-06-01       0.983853
dtype: float64
```

Random S&P 500 prices (2)

```
data['SP500_random'] = sp500_random.cumprod()
data[['SP500', 'SP500_random']].plot()
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Relationships between time series: correlation

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

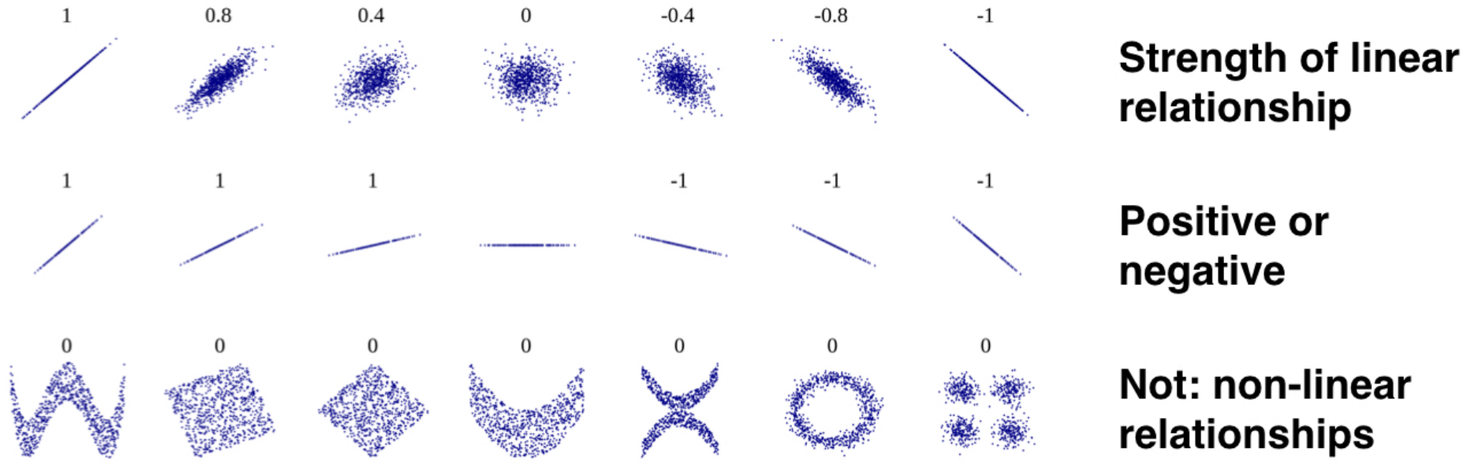
Correlation & relations between series

- So far, focus on characteristics of individual variables
- Now: characteristic of relations between variables
- Correlation: measures linear relationships
- Financial markets: important for prediction and risk management
- `pandas` & `seaborn` have tools to compute & visualize

Correlation & linear relationships

- Correlation coefficient: how similar is the pairwise movement of two variables around their averages?
- Varies between `-1` and `+1`

$$r = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{s_x s_y}$$



Strength of linear relationship

Positive or negative

Not: non-linear relationships

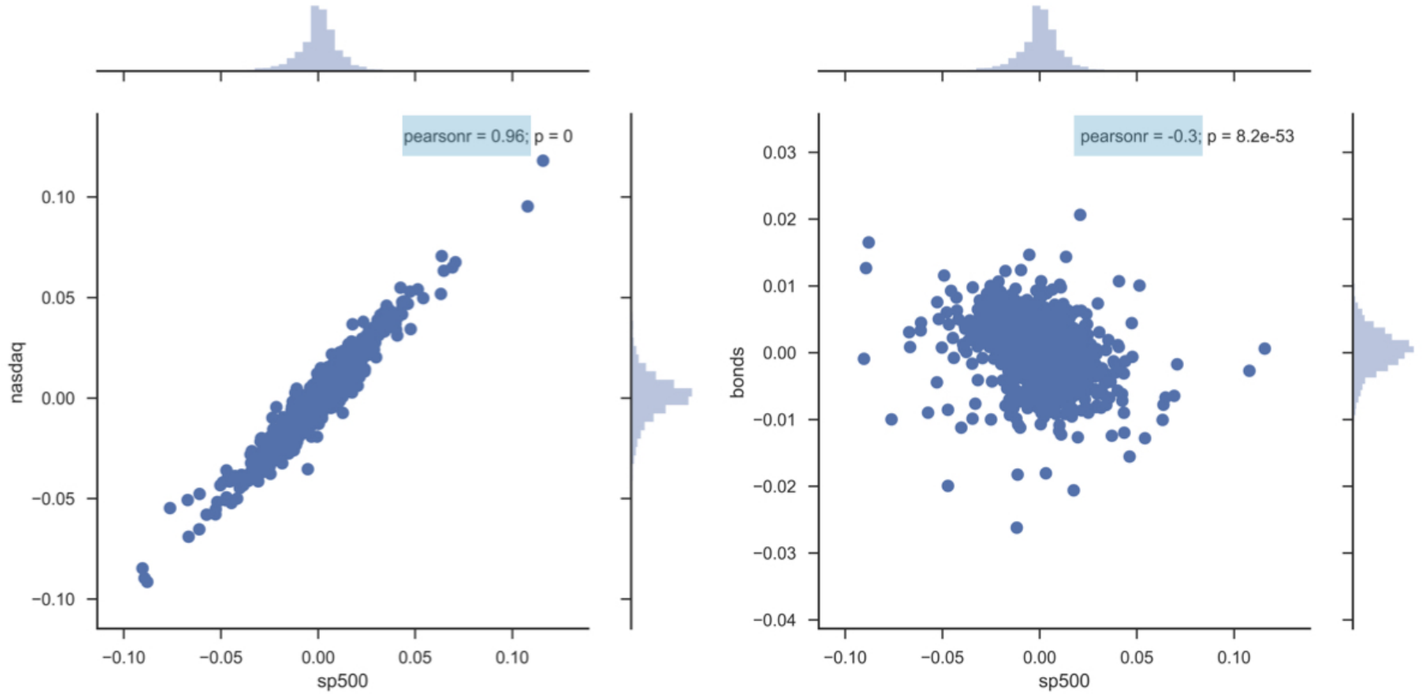
Importing five price time series

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

```
DatetimeIndex: 2469 entries, 2007-05-25 to 2017-05-22
Data columns (total 5 columns):
sp500     2469 non-null float64
nasdaq    2469 non-null float64
bonds     2469 non-null float64
gold      2469 non-null float64
oil       2469 non-null float64
```

Visualize pairwise linear relationships

```
daily_returns = data.pct_change()
sns.jointplot(x='sp500', y='nasdaq', data=daily_returns);
```



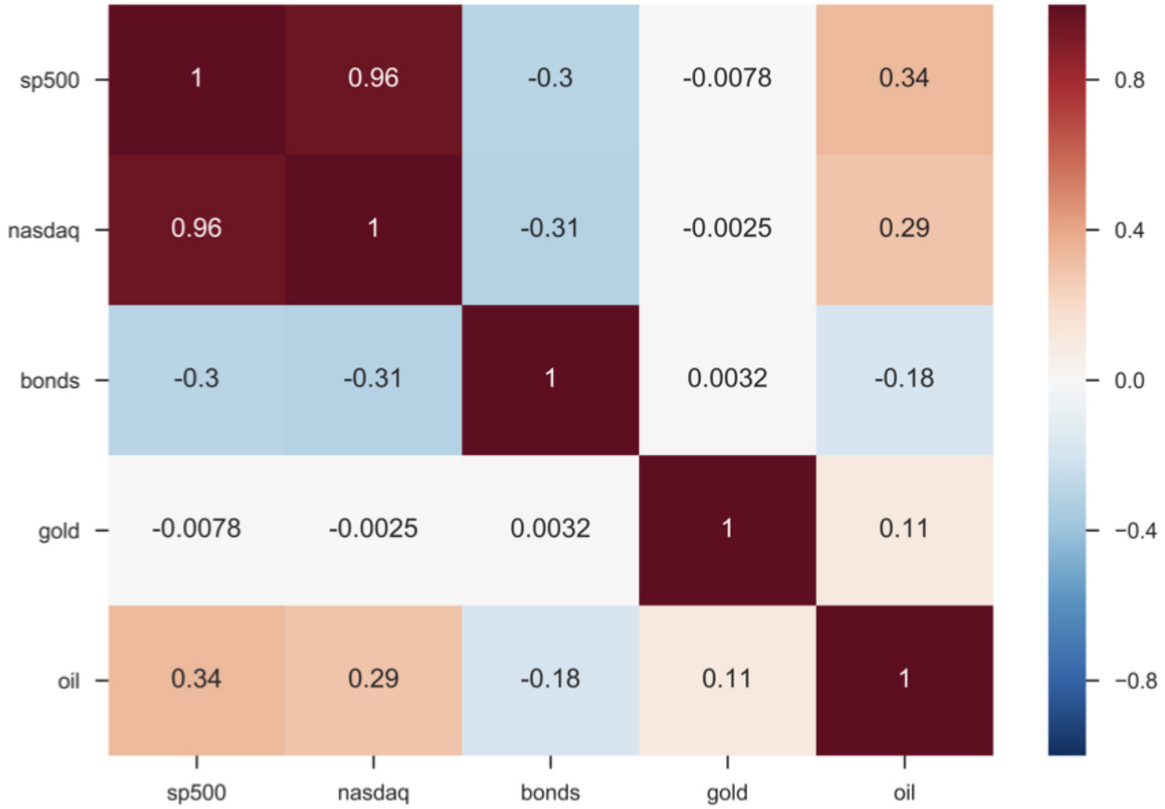
Calculate all correlations

```
correlations = returns.corr()
correlations
```

```
bonds       oil      gold     sp500    nasdaq
bonds   1.000000 -0.183755  0.003167 -0.300877 -0.306437
oil    -0.183755  1.000000  0.105930  0.335578  0.289590
gold    0.003167  0.105930  1.000000 -0.007786 -0.002544
sp500  -0.300877  0.335578 -0.007786  1.000000  0.959990
nasdaq -0.306437  0.289590 -0.002544  0.959990  1.000000
```

Visualize all correlations

```
sns.heatmap(correlations, annot=True)
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Select index components & import data

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Market value-weighted index

- Composite performance of various stocks
- Components weighted by market capitalization
  - `Share Price` x `Number of Shares` => `Market Value`
- Larger components get higher percentage weightings
- Key market indexes are value-weighted:
  - `S&P 500`, `NASDAQ`, `Wilshire 5000`, `Hang Seng`

Build a cap-weighted Index

- Apply new skills to construct value-weighted index
  - Select components from exchange listing data
  - Get component number of shares and stock prices
  - Calculate component weights
  - Calculate index
  - Evaluate performance of components and index

Load stock listing data

```
nyse = pd.read_excel('listings.xlsx', sheet_name='nyse',
                     na_values='n/a')
nyse.info()
```

```
RangeIndex: 3147 entries, 0 to 3146
Data columns (total 7 columns):
Stock Symbol             3147 non-null object   # Stock Ticker
Company Name             3147 non-null object
Last Sale                3079 non-null float64  # Latest Stock Price
Market Capitalization    3147 non-null float64
IPO Year                 1361 non-null float64  # Year of listing
Sector                   2177 non-null object
Industry                 2177 non-null object
dtypes: float64(3), object(4)
```

Load & prepare listing data

```
nyse.set_index('Stock Symbol', inplace=True)
nyse.dropna(subset=['Sector'], inplace=True)
nyse['Market Capitalization'] /= 1e6  # in Million USD
```

```
Index: 2177 entries, DDD to ZTO
Data columns (total 6 columns):
Company Name              2177 non-null object
Last Sale                 2175 non-null float64
Market Capitalization     2177 non-null float64
IPO Year                  967 non-null float64
Sector                    2177 non-null object
Industry                  2177 non-null object
dtypes: float64(3), object(3)
```

Select index components

```
components = nyse.groupby(['Sector'])['Market Capitalization'].nlargest(1)
components.sort_values(ascending=False)
```

```
Sector                  Stock Symbol
Health Care             JNJ             338834.390080
Energy                  XOM             338728.713874
Finance                 JPM             300283.250479
Miscellaneous           BABA            275525.000000
Public Utilities        T               247339.517272
Basic Industries        PG              230159.644117
Consumer Services       WMT             221864.614129
Consumer Non-Durables   KO              183655.305119
Technology              ORCL            181046.096000
Capital Goods           TM              155660.252483
Transportation          UPS              90180.886756
Consumer Durables       ABB              48398.935676
Name: Market Capitalization, dtype: float64
```

Import & prepare listing data

```
tickers = components.index.get_level_values('Stock Symbol')
tickers
```

```
Index(['PG', 'TM', 'ABB', 'KO', 'WMT', 'XOM', 'JPM', 'JNJ', 'BABA', 'T',
       'ORCL', ‘UPS'], dtype='object', name='Stock Symbol’)
```

```
tickers.tolist()
```

```
['PG',
 'TM',
 'ABB',
 'KO',
 'WMT',
...
 'T',
 'ORCL',
 'UPS']
```

Stock index components

```
columns = ['Company Name', 'Market Capitalization', 'Last Sale']
component_info = nyse.loc[tickers, columns]
pd.options.display.float_format = '{:,.2f}'.format
```

Stock Symbol	Company Name	Market Capitalization	Last Sale
PG	Procter & Gamble Company (The)	230,159.64	90.03
TM	Toyota Motor Corp Ltd Ord	155,660.25	104.18
ABB	ABB Ltd	48,398.94	22.63
KO	Coca-Cola Company (The)	183,655.31	42.79
WMT	Wal-Mart Stores, Inc.	221,864.61	73.15
XOM	Exxon Mobil Corporation	338,728.71	81.69
JPM	J P Morgan Chase & Co	300,283.25	84.40
JNJ	Johnson & Johnson	338,834.39	124.99
BABA	Alibaba Group Holding Limited	275,525.00	110.21
T	AT&T Inc.	247,339.52	40.28
ORCL	Oracle Corporation	181,046.10	44.00
UPS	United Parcel Service, Inc.	90,180.89	103.74

Import & prepare listing data

```
data = pd.read_csv('stocks.csv', parse_dates=['Date'],
                                 index_col='Date').loc[:, tickers.tolist()]
data.info()
```

```
DatetimeIndex: 252 entries, 2016-01-04 to 2016-12-30
Data columns (total 12 columns):
ABB     252 non-null float64
BABA    252 non-null float64
JNJ     252 non-null float64
JPM     252 non-null float64
KO      252 non-null float64
ORCL    252 non-null float64
PG      252 non-null float64
T       252 non-null float64
TM      252 non-null float64
UPS     252 non-null float64
WMT     252 non-null float64
XOM     252 non-null float64
dtypes: float64(12)
```

Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Build a market-cap weighted index

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Build your value-weighted index

- Key inputs:
  - number of shares
  - stock price series

Build your value-weighted index

- Key inputs:
  - number of shares
  - stock price series
  - Normalize index to start at 100

Stock index components

```
components
```

Stock Symbol	Company Name	Market Capitalization	Last Sale
PG	Procter & Gamble Company (The)	230,159.64	90.03
TM	Toyota Motor Corp Ltd Ord	155,660.25	104.18
ABB	ABB Ltd	48,398.94	22.63
KO	Coca-Cola Company (The)	183,655.31	42.79
WMT	Wal-Mart Stores, Inc.	221,864.61	73.15
XOM	Exxon Mobil Corporation	338,728.71	81.69
JPM	J P Morgan Chase & Co	300,283.25	84.40
JNJ	Johnson & Johnson	338,834.39	124.99
BABA	Alibaba Group Holding Limited	275,525.00	110.21
T	AT&T Inc.	247,339.52	40.28
ORCL	Oracle Corporation	181,046.10	44.00
UPS	United Parcel Service, Inc.	90,180.89	103.74

Number of shares outstanding

```
shares = components['Market Capitalization'].div(components['Last Sale'])
```

```
Stock Symbol
PG     2,556.48 # Outstanding shares in million
TM     1,494.15
ABB    2,138.71
KO     4,292.01
WMT    3,033.01
XOM    4,146.51
JPM    3,557.86
JNJ    2,710.89
BABA   2,500.00
T      6,140.50
ORCL   4,114.68
UPS      869.30
dtype: float64
```

- Market Capitalization = Number of Shares x Share Price

Historical stock prices

```
data = pd.read_csv('stocks.csv', parse_dates=['Date'],
                           index_col='Date').loc[:, tickers.tolist()]
market_cap_series = data.mul(no_shares)
market_series.info()
```

```
DatetimeIndex: 252 entries, 2016-01-04 to 2016-12-30
Data columns (total 12 columns):
ABB     252 non-null float64
BABA    252 non-null float64
JNJ     252 non-null float64
JPM     252 non-null float64
...
TM      252 non-null float64
UPS     252 non-null float64
WMT     252 non-null float64
XOM     252 non-null float64
dtypes: float64(12)
```

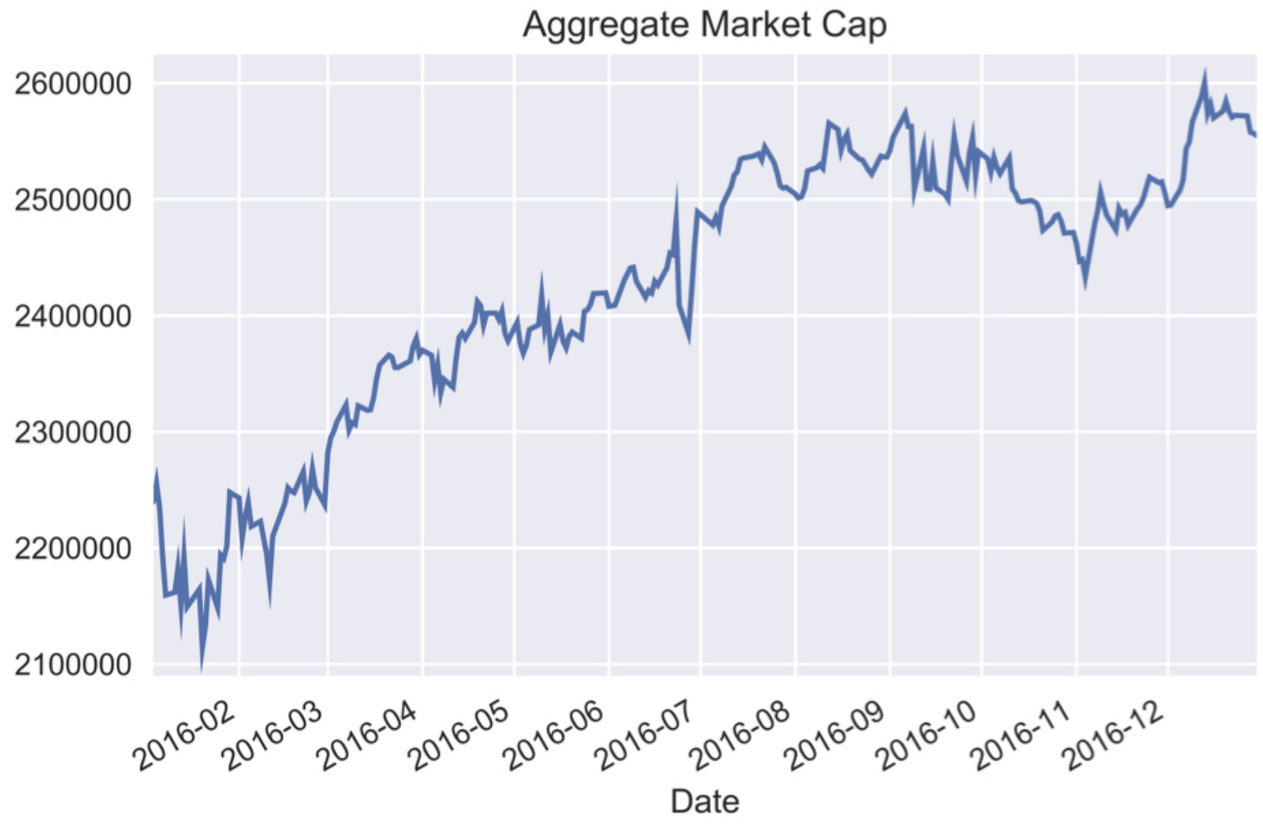
From stock prices to market value

```
market_cap_series.first('D').append(market_cap_series.last('D'))
```

```
                ABB       BABA        JNJ        JPM         KO        ORCL  \\
Date
2016-01-04 37,470.14 191,725.00 272,390.43 226,350.95 181,981.42 147,099.95
2016-12-30 45,062.55 219,525.00 312,321.87 307,007.60 177,946.93 158,209.60
                  PG          T         TM       UPS        WMT         XOM
Date
2016-01-04 200,351.12 210,926.33 181,479.12 82,444.14 186,408.74 321,188.96
2016-12-30 214,948.60 261,155.65 175,114.05 99,656.23 209,641.59 374,264.34
```

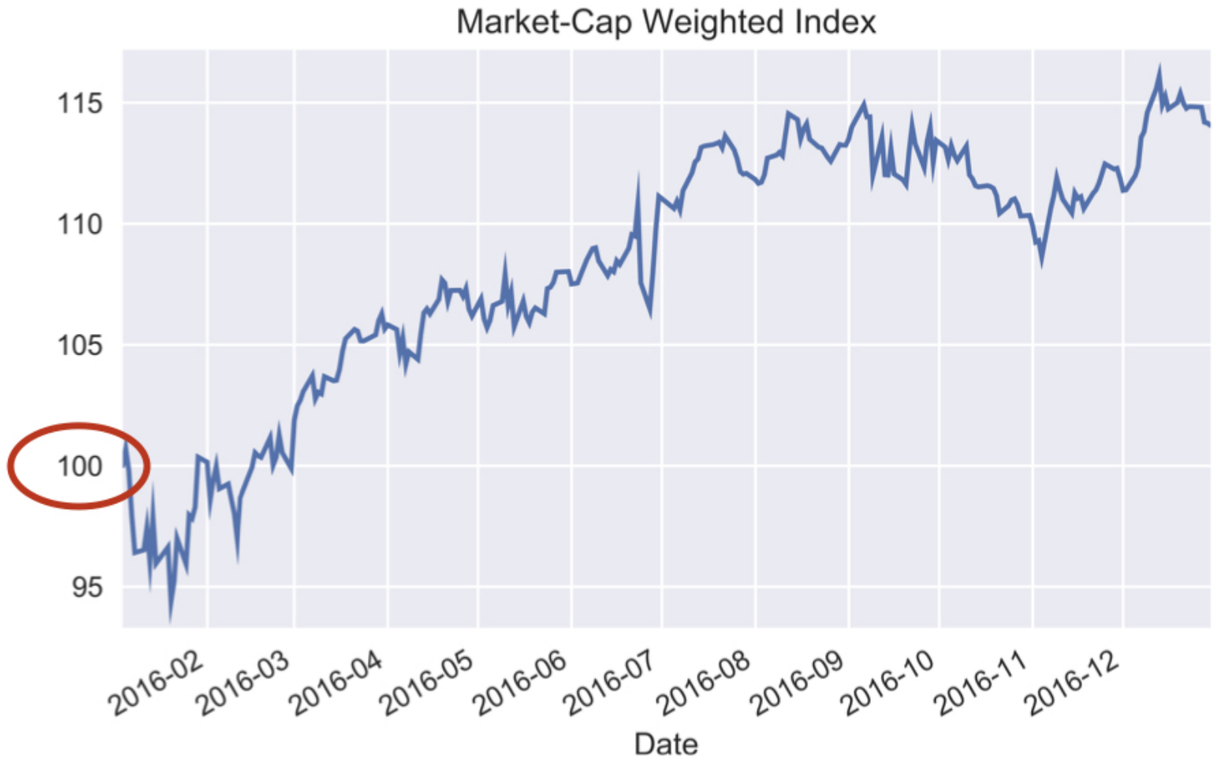
Aggregate market value per period

```
agg_mcap = market_cap_series.sum(axis=1) # Total market cap
agg_mcap(title='Aggregate Market Cap')
```



Value-based index

```
index = agg_mcap.div(agg_mcap.iloc[0]).mul(100) # Divide by 1st value
index.plot(title='Market-Cap Weighted Index')
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Evaluate index performance

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

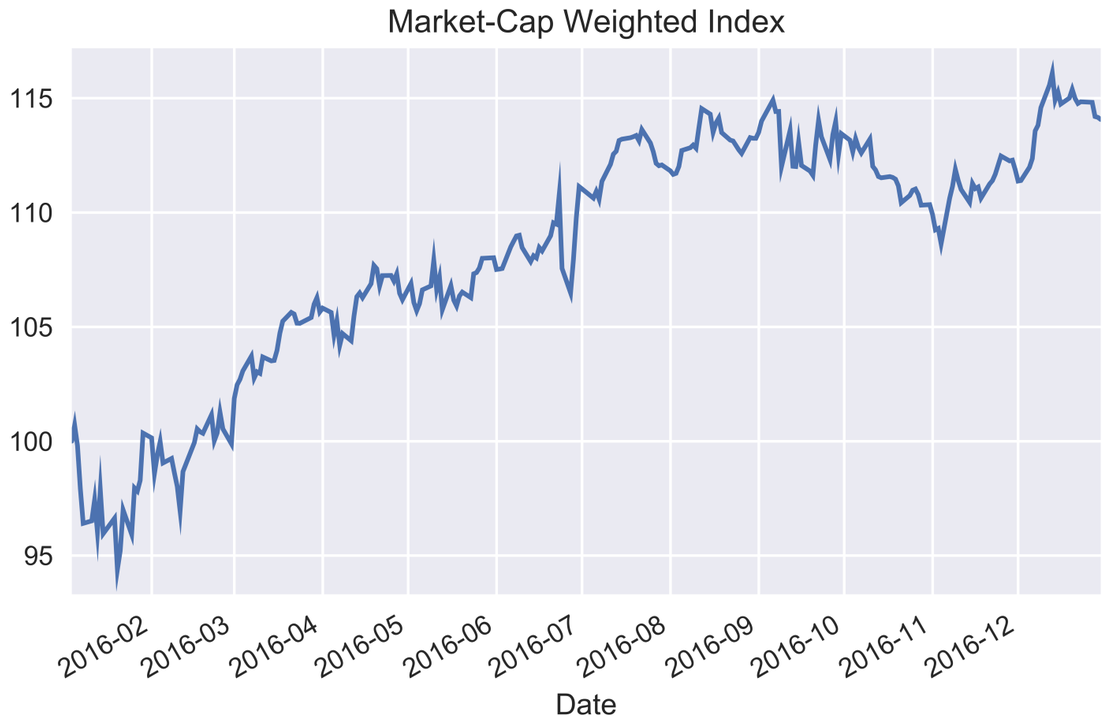
datacamp

Evaluate your value-weighted index

- Index return:
  - Total index return
  - Contribution by component
- Performance vs Benchmark
  - Total period return
  - Rolling returns for sub periods

Value-based index - recap

```
agg_market_cap = market_cap_series.sum(axis=1)
index = agg_market_cap.div(agg_market_cap.iloc[0]).mul(100)
index.plot(title='Market-Cap Weighted Index')
```



Value contribution by stock

```
agg_market_cap.iloc[-1] - agg_market_cap.iloc[0]
```

```
315,037.71
```

Value contribution by stock

```
change = market_cap_series.first('D').append(market_cap_series.last('D'))
change.diff().iloc[-1].sort_values() # or: .loc['2016-12-30']
```

```
TM     -6,365.07
KO     -4,034.49
ABB     7,592.41
ORCL   11,109.65
PG     14,597.48
UPS    17,212.08
WMT    23,232.85
BABA   27,800.00
JNJ    39,931.44
T      50,229.33
XOM    53,075.38
JPM    80,656.65
Name: 2016-12-30 00:00:00, dtype: float64
```

Market-cap based weights

```
market_cap = components['Market Capitalization']
weights = market_cap.div(market_cap.sum())
weights.sort_values().mul(100)
```

```
Stock Symbol
ABB     1.85
UPS     3.45
TM      5.96
ORCL    6.93
KO      7.03
WMT     8.50
PG      8.81
T       9.47
BABA   10.55
JPM    11.50
XOM    12.97
JNJ    12.97
Name: Market Capitalization, dtype: float64
```

Value-weighted component returns

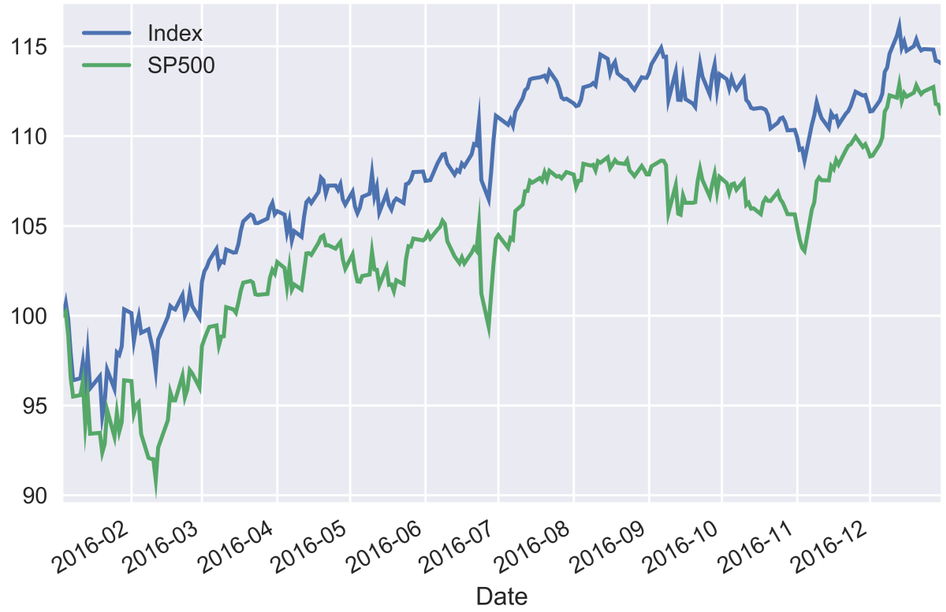
```
index_return = (index.iloc[-1] / index.iloc[0] - 1) * 100
```

```
14.06
```

```
weighted_returns = weights.mul(index_return)
weighted_returns.sort_values().plot(kind='barh')
```

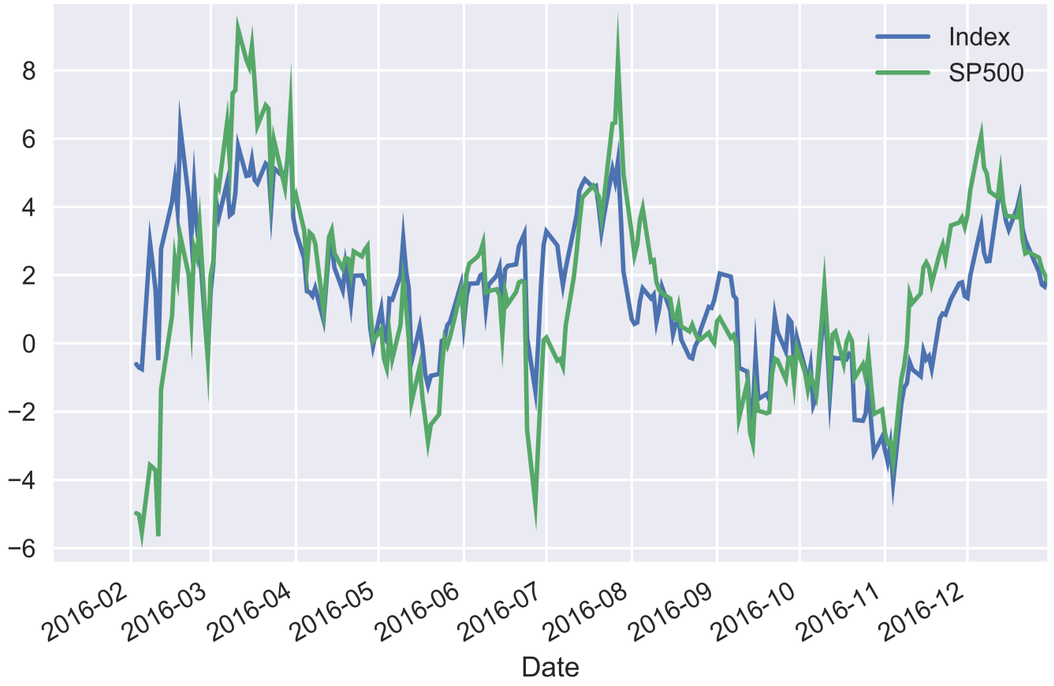
Performance vs benchmark

```
data = index.to_frame('Index') # Convert pd.Series to pd.DataFrame
data['SP500'] = pd.read_csv('sp500.csv', parse_dates=['Date'],
                            index_col='Date')
data.SP500 = data.SP500.div(data.SP500.iloc[0], axis=0).mul(100)
```



Performance vs benchmark: 30D rolling return

```
def multi_period_return(r):
    return (np.prod(r + 1) - 1) * 100
data.pct_change().rolling('30D').apply(multi_period_return).plot()
```



Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Index correlation & exporting to Excel

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

datacamp

Some additional analysis of your index

- Daily return correlations:
- Calculate among all components
- Visualize the result as heatmap
- Write results to excel using `.xls` and `.xlsx` formats:
- Single worksheet
- Multiple worksheets

Index components - price data

```
data = DataReader(tickers, 'google', start='2016', end='2017')['Close']
data.info()
```

```
DatetimeIndex: 252 entries, 2016-01-04 to 2016-12-30
Data columns (total 12 columns):
ABB     252 non-null float64
BABA    252 non-null float64
JNJ     252 non-null float64
JPM     252 non-null float64
KO      252 non-null float64
ORCL    252 non-null float64
PG      252 non-null float64
T       252 non-null float64
TM      252 non-null float64
UPS     252 non-null float64
WMT     252 non-null float64
XOM     252 non-null float64
```

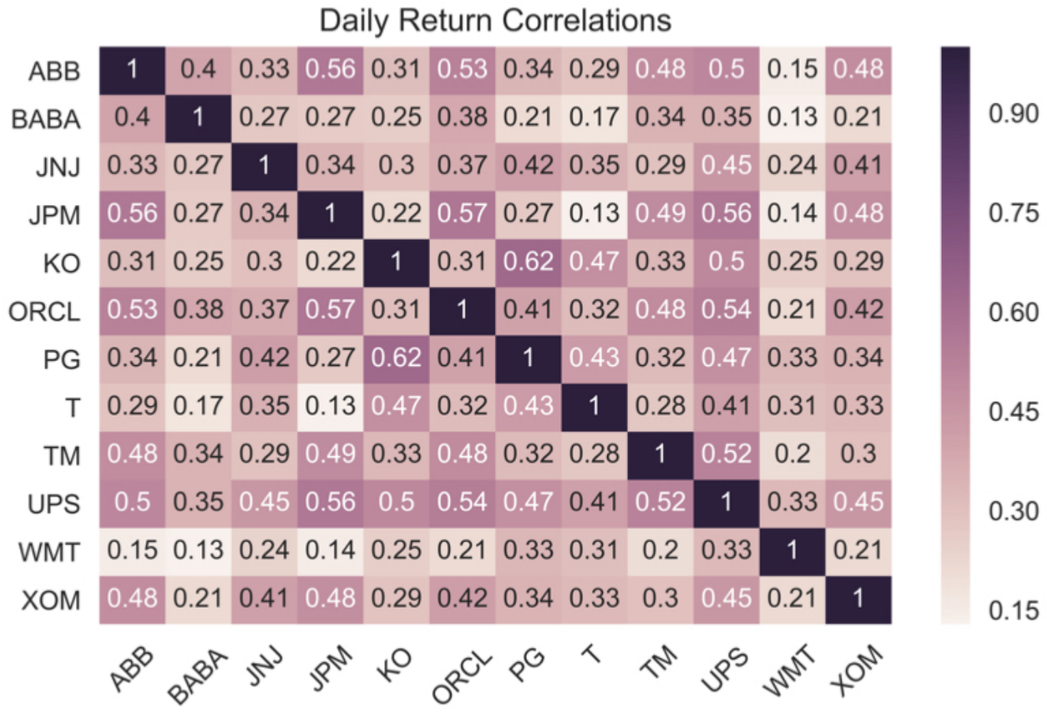
Index components: return correlations

```
daily_returns = data.pct_change()
correlations = daily_returns.corr()
```

```
ABB  BABA  JNJ  JPM   KO  ORCL   PG    T   TM  UPS  WMT  XOM
ABB  1.00  0.40 0.33 0.56 0.31  0.53 0.34 0.29 0.48 0.50 0.15 0.48
BABA 0.40  1.00 0.27 0.27 0.25  0.38 0.21 0.17 0.34 0.35 0.13 0.21
JNJ  0.33  0.27 1.00 0.34 0.30  0.37 0.42 0.35 0.29 0.45 0.24 0.41
JPM  0.56  0.27 0.34 1.00 0.22  0.57 0.27 0.13 0.49 0.56 0.14 0.48
KO   0.31  0.25 0.30 0.22 1.00  0.31 0.62 0.47 0.33 0.50 0.25 0.29
ORCL 0.53  0.38 0.37 0.57 0.31  1.00 0.41 0.32 0.48 0.54 0.21 0.42
PG   0.34  0.21 0.42 0.27 0.62  0.41 1.00 0.43 0.32 0.47 0.33 0.34
T    0.29  0.17 0.35 0.13 0.47  0.32 0.43 1.00 0.28 0.41 0.31 0.33
TM   0.48  0.34 0.29 0.49 0.33  0.48 0.32 0.28 1.00 0.52 0.20 0.30
UPS  0.50  0.35 0.45 0.56 0.50  0.54 0.47 0.41 0.52 1.00 0.33 0.45
WMT  0.15  0.13 0.24 0.14 0.25  0.21 0.33 0.31 0.20 0.33 1.00 0.21
XOM  0.48  0.21 0.41 0.48 0.29  0.42 0.34 0.33 0.30 0.45 0.21 1.00
```

Index components: return correlations

```
sns.heatmap(correlations, annot=True)
plt.xticks(rotation=45)
plt.title('Daily Return Correlations')
```



Saving to a single Excel worksheet

```
correlations.to_excel(excel_writer= 'correlations.xls',
                      sheet_name='correlations',
                      startrow=1,
                      startcol=1)
```

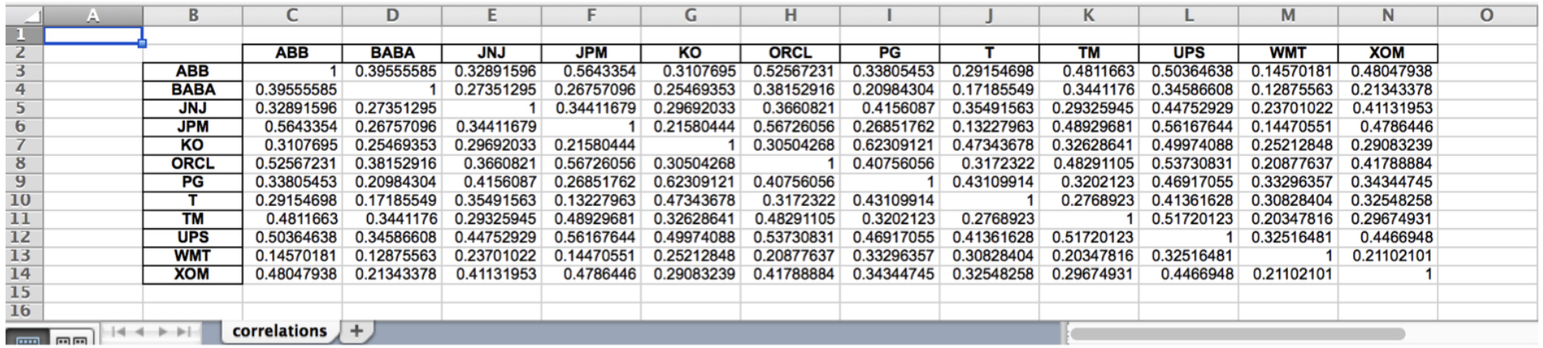


	ABB	BABA	JNJ	JPM	KO	ORCL	PG	T	TM	UPS	WMT	XOM
ABB	1	0.39555585	0.32891596	0.5643354	0.3107695	0.52567231	0.33805453	0.29154698	0.4811663	0.50364638	0.14570181	0.48047938
BABA	0.39555585	1	0.27351295	0.26757096	0.25469353	0.38152916	0.20984304	0.17185549	0.3441176	0.34586608	0.12875563	0.21343378
JNJ	0.32891596	0.27351295	1	0.34411679	0.29692033	0.3660821	0.4156087	0.35491563	0.29325945	0.44752929	0.23701022	0.41131953
JPM	0.5643354	0.26757096	0.34411679	1	0.21580444	0.56726056	0.26851762	0.13227963	0.48929681	0.56167644	0.14470551	0.4786446
KO	0.3107695	0.25469353	0.29692033	0.21580444	1	0.30504268	0.62309121	0.47343678	0.32628641	0.49974088	0.25212848	0.29083239
ORCL	0.52567231	0.38152916	0.3660821	0.56726056	0.30504268	1	0.40756056	0.3172322	0.48291105	0.53730831	0.20877637	0.41788884
PG	0.33805453	0.20984304	0.4156087	0.26851762	0.62309121	0.40756056	1	0.43109914	0.3202123	0.46917055	0.33296357	0.34344745
T	0.29154698	0.17185549	0.35491563	0.13227963	0.47343678	0.3172322	0.43109914	1	0.2768923	0.41361628	0.30828404	0.32548258
TM	0.4811663	0.3441176	0.29325945	0.48929681	0.32628641	0.48291105	0.3202123	0.2768923	1	0.51720123	0.20347816	0.29674931
UPS	0.50364638	0.34586608	0.44752929	0.56167644	0.49974088	0.53730831	0.46917055	0.41361628	0.51720123	1	0.32516481	0.4466948
WMT	0.14570181	0.12875563	0.23701022	0.14470551	0.25212848	0.20877637	0.33296357	0.30828404	0.20347816	0.32516481	1	0.21102101
XOM	0.48047938	0.21343378	0.41131953	0.4786446	0.29083239	0.41788884	0.34344745	0.32548258	0.29674931	0.4466948	0.21102101	1

Saving to multiple Excel worksheets

```
data.index = data.index.date # Keep only date component
with pd.ExcelWriter('stock_data.xlsx') as writer:
    corr.to_excel(excel_writer=writer, sheet_name='correlations')
    data.to_excel(excel_writer=writer, sheet_name='prices')
    data.pct_change().to_excel(writer, sheet_name='returns')
```

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		ABB	BABA	JNJ	JPM	KO	ORCL	PG	T	TM	UPS	WMT	XOM
2	2016-01-04	17.52	76.69	100.48	63.62	42.4	35.75	78.37	34.35	121.46	94.84	61.46	77.46
3	2016-01-05	17.21	78.63	100.9	63.73	42.55	35.64	78.62	34.59	121.14	95.78	62.92	78.12
4	2016-01-06	16.92	77.33	100.39	62.81	42.32	35.82	77.86	34.06	118.38	94.42	63.55	77.47
5	2016-01-07	16.6	72.72	99.22	60.27	41.62	35.04	77.18	33.51	115.57	92.6	65.03	76.23
6	2016-01-08	16.31	70.8	98.16	58.92	41.51	34.65	75.97	33.54	113.06	91.39	63.54	74.69
7	2016-01-11	16.31	69.92	97.57	58.83	41.58	34.94	76.67	33.95	114.81	91.66	64.22	73.69
8	2016-01-12	16.66	72.68	98.24	58.96	42.12	35.37	76.51	33.9	115.83	93	63.62	75.2
9	2016-01-13	16.3	70.29	97.02	57.34	41.85	34.08	75.85	33.74	114.99	90.61	61.92	75.65
10	2016-01-14	16.73	72.25	98.89	58.2	41.88	34.79	76.15	34.3	116.29	91.15	63.06	79.12
11	2016-01-15	16.06	69.59	97	57.04	41.5	34.12	74.98	33.99	112.6	90.04	61.93	77.58
12	2016-01-19	16.26	70.13	97.5	57.01	41.92	34.55	76.73	34.51	115.02	90.35	62.56	76.4

correlations | prices | returns | +

Let's practice!

MANIPULATING TIME SERIES DATA IN PYTHON

Congratulations!

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at
Applied Artificial Intelligence

Congratulations!

MANIPULATING TIME SERIES DATA IN PYTHON