Data Camp Various Python Courses

# Manipulating Time Series Data in Python

## Working with Time Series in Pandas

### How to use dates and times with pandas

TimeStamp in pandas replaces python datetime.datetime object. It adds properties such as frequency.

data.date = pd.to\_datetime(data.date)

data.set\_index("date", inplace=True)

It converts the date column to type datetime64[ns] and creates the DatetimeIndex column

type(data.index)

> <class 'pandas.core.indexes.datetimes.DatetimeIndex'>

type(data.index[0])

> <class 'pandas.\_libs.tslibs.timestamps.Timestamp'>

DatetimeIndex consists of Timestamps

pd.date\_range(start, end, periods, freq)

creates a sequence of Timestamp objects with frequency info

### Indexing and resampling time series

We can use different ways to filter and index a time series

google[“2021”]

google[“2021-6”: “2023-6”]

google.loc[“2021-6”: “2023-6”, [“open”, “close”]]

Use pandas.dataframe.asfreq() method to add frequency to your Datetime index.

#### Up sampling & down sampling

Up sampling adds NA for missing values. Down sampling aggregates the observations.

### Lags, changes and returns for stock price series

**shift()** shifts days into the future or back in time

**diff()** calculates difference in value for two adjacent periods Xt – Xt-1

**pct\_change()** calculates the return in percentage change on a period Xt / Xt-1. It can also be calculated manually as “return1”

yahoo["shifted"] = yahoo["price"].shift()

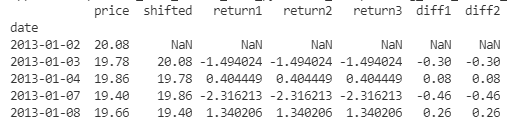
yahoo["return1"] = yahoo["price"].div(yahoo["shifted"]).sub(1).mul(100)

yahoo["return2"] = (yahoo["price"] / yahoo["shifted"] - 1) \* 100

yahoo["return3"] = yahoo["price"].pct\_change().mul(100)

yahoo["diff1"] = yahoo["price"].sub(yahoo["shifted"])

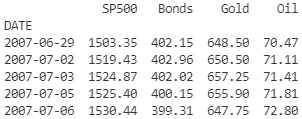
yahoo["diff2"] = yahoo["price"].diff()



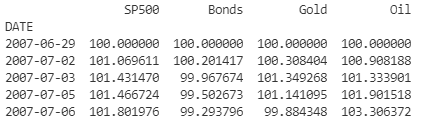
## Basic Time Series Metrics and Resampling

### Compare time series growth rates

In order to compare stock prices from different levels, we normalize the price series to start at 100



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



### To compare asset classes to a bench mark index

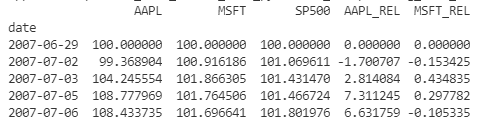
Treat the benchmark index as one of the asset classes, normalized all asset classes’ price series to start at 100

### Compare performance difference vs benchmark index

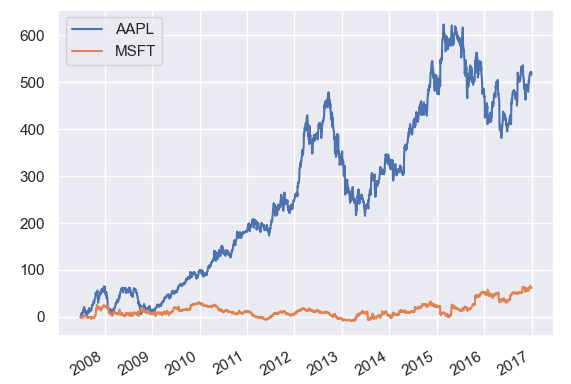
Normalize all asset classes (including index) to start at 100.   
Subtract index from asset classes.

normalized[["MSFT", "AAPL"]].sub(normalized["SP500"], axis=0)

We get MSFT and AAPL’s performances relative to SP500



In a 9 year period, AAPL performances 5 times better than SP500.



### Changing the time series frequency: resampling

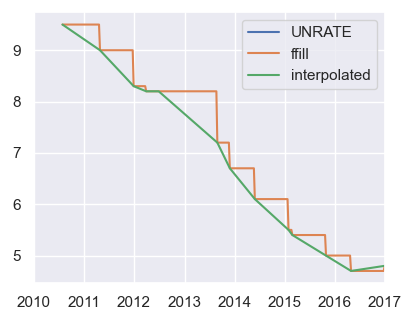
asfreq() resamples the datetime sequence to specified frequency. Depending on the nature of the operations, when up sampling, use fill\_value, method= “ffill” / “bfill” to fill the missing value. When downsampling, specify the aggregation method.

### Up sampling and interpolation

Interpolate interpolates the data points in between the actual ones.

weekly['ffill'] = weekly.UNRATE.ffill()

weekly['interpolated'] = weekly.UNRATE.interpolate()



### Down sampling and aggregation

stocks.resample("M").mean()

daily\_returns.resample("M").agg(["mean", "median", "std"])

## Window Functions: Rolling and Expanding Metrics

What are window functions:

* 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
  + Expanding: contain all prior values

### Rolling Window

We can use rolling window to compute the moving average to get a smoother curve.

data["Ozone"].rolling(window="90D").mean()

Or we can plot an upper and lower band using standard deviations

data['q10'] = rolling.quantile(0.1).to\_frame("q10")

data['q90'] = rolling.quantile(0.9).to\_frame("q90")

### Expanding Window

What are expanding windows:

* Calculate metrics for periods up to current date
* Creates new time series that reflect all historical values
* Useful for running rate of return, running min/max
* Two options with pandas
  + .expend() just like .rolling()
  + .cumsum(), cumprod(), cumin() cummax()

How to calculate a running return

**Single period return** Rt: current price over last price minus 1

Rt = (Pt / Pt-1) – 1

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

Rt = (1 + R1) \* (1 + R2) \* … \* (1 + Rt) - 1

In code, we implement it as

# Calculate the daily returns here

returns = data.pct\_change()

# Calculate the cumulative returns here

returns\_plus\_one = returns + 1

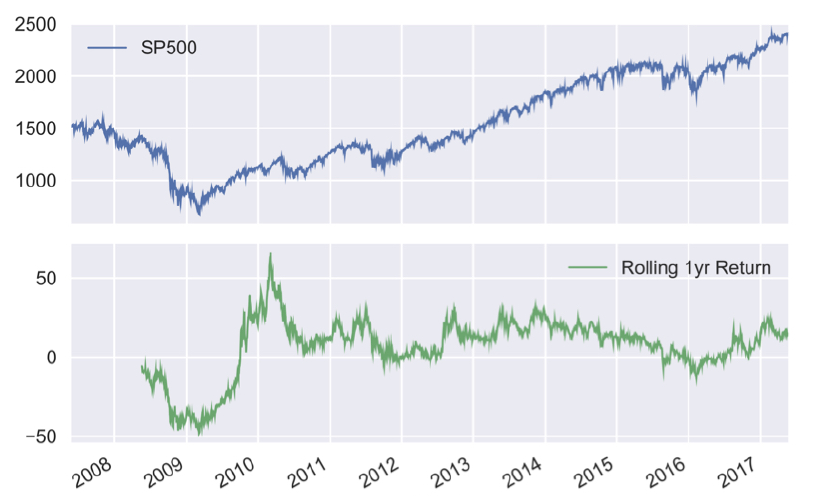
cumulative\_return = returns\_plus\_one.cumprod()

#### Rolling annual rate of return

def multi\_period\_return(period\_returns):

    return np.prod(period\_returns + 1) - 1

rolling\_annual\_returns = daily\_returns.rolling("360D").apply(multi\_period\_return)



### SP500 Price Simulation

Random walk theory & simulations

* Daily stock returns are hard to predict
* Models often assume they are random in nature

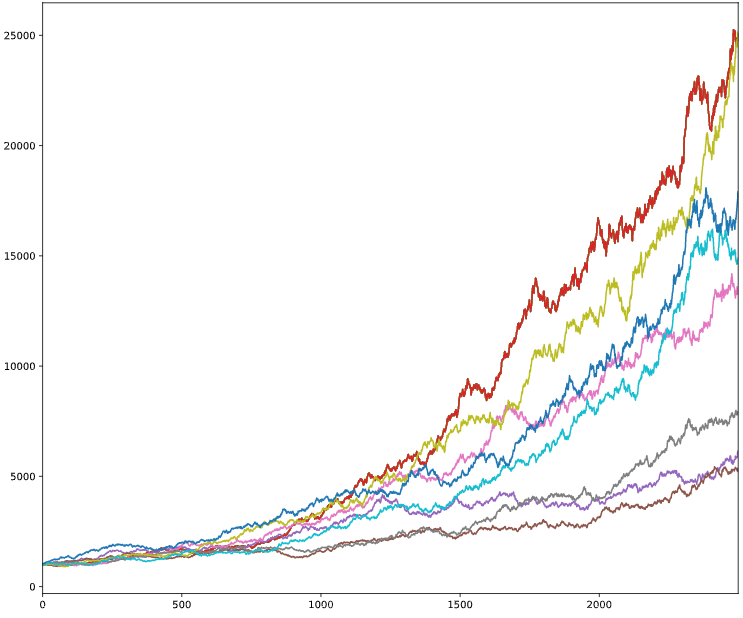
#### Example 1 - Generate random returns

random\_walk = normal(loc=.001, scale=.01, size=2500)

random\_prices = random\_walk.add(1).cumprod()

random\_prices.mul(1000).plot()

As we can see, an average daily return of merely 0.1% compounds into a high flying chart



#### Example 2 – Random selected actual SP500 returns

We can randomly select historical SP500 returns and simulate its price action

daily\_returns = fb.price.pct\_change().dropna()

random\_walk = np.random.choice(daily\_returns, n\_obs)

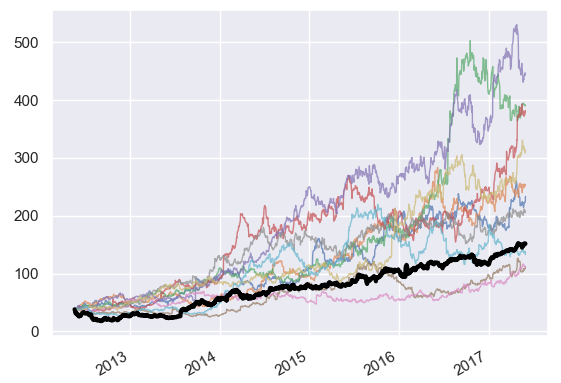
start = fb.price.first('D')

random\_walk = random\_walk.add(1)

random\_price = start.append(random\_walk)

random\_price = random\_price.cumprod()

If the actual price has large deviations, the random walk can produce result that are very different from the actual observation. Because large positives returns or large negative returns can compound in consecutive days.



### Correlations between Time Series

Correaltions ought to be computed on returns, not on the price itself. As two rising stock prices will always have high correlation. Use seaborn heatmap to visualize the correlation.

annual\_returns = annual\_prices.pct\_change()

correlations = annual\_returns.corr()

sns.heatmap(correlations, annot=True)

## Building a value-weighted index

What is a market value-weighted index?

* A stock index is a composition of various stocks
* Components weighted by market capitalization  
   share price \* number of shares = market value

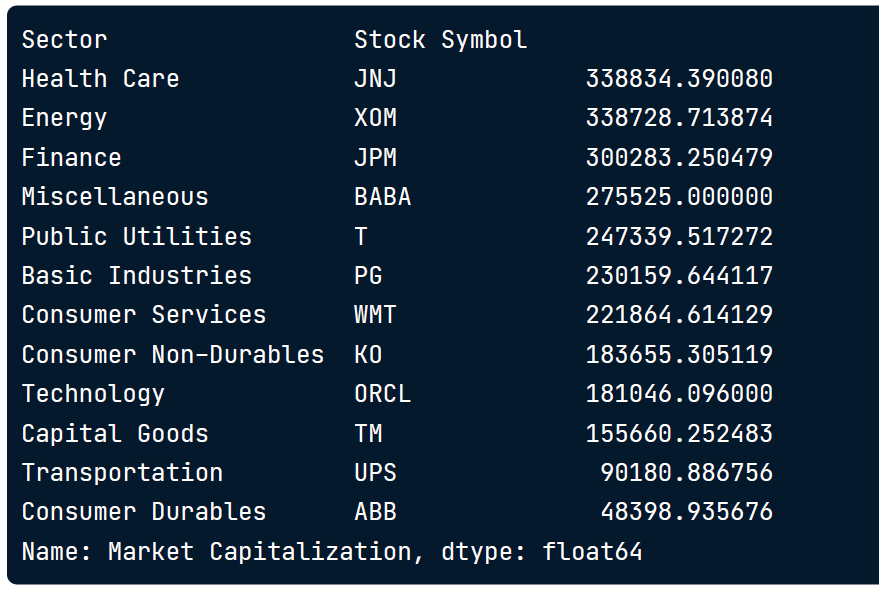
Steps of building a cap-weighted index

* 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

### Selecting Index Components

Selecting stocks with largest market capitalization from each sector

components = listings.groupby("Sector")["Market Capitalization"].nlargest(1)



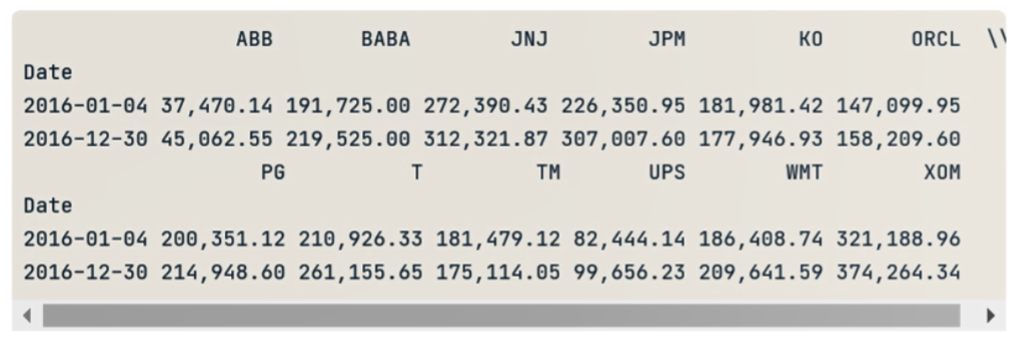
### Build a Market-cap weighted Index

Calculate number of shares by dividing the market capitalization by last sale price

no\_shares = components["Market Capitalization"] / components["Last Sale"]

Calculate aggregated market value per period

market\_cap = stock\_prices.mul(no\_shares)



Sum all the columns to one to get the aggregated index value

raw\_index = market\_cap\_series.sum(axis=1)

Normalize the raw index to start at 100

index = raw\_index.div(raw\_index.iloc[0]).mul(100)

### Evaluate Index Performance

We can evaluate the index in the following metrics

* Index return:
  + Total period return
  + Contribution by components
* Performance vs Benchmark
  + Total period return
  + Rolling returns for sub periods

### Index Correlation

We can analyze the daily return correlations of the index components

# Time Series Analysis in Python

## Correlation and Autocorrelation

## Some Simple Time Series

## Autoregressive (AR) Models

## Moving Average (MA) and ARMA Models

## Putting it All Together

# Visualizing Time Series Data in Python

# ARIMA Models in Python

# Machine Learning for Time Series Data in Python