Time Series Analysis

September 9, 2020

Handling Time series with Pandas and manipulating Financial Data

2019DMB02 - Akash Gupta

Here is a very basic introduction to the fundamental building blocks to time series data representations in Python. There are packages that offer efficient computation on time series data and can be used to analyze financial data as well, as has been demonstrated at the end of this note using **Google** stock.

[5]:
```
import pandas as pd
import numpy as np
```

Date and time data can come in these formats:

1. **Time stamps** - references particular moments in time (July 15th, 2016, 7:00am).
2. **Time intervals** - references a length of time between a particular beginning and end point (the year 2015).
3. **Time delta** - an exact length of time (22.58 seconds).

In Python we work with time series data using the **date-time** and **dateutil** module. Below it is shown how we can manually build a date using the datetime module.

[22]:
```
from datetime import datetime
datetime(year=2015, month=7, day=4)
```

[22]:
```
datetime.datetime(2015, 7, 4, 0, 0)
```

We can use **dateutil** module we can parse or extract the numeric datetime format from a text based date.

[3]:
```
from dateutil import parser
date = parser.parse("4th of July, 2015")
date
```

[3]:
```
datetime.datetime(2015, 7, 4, 0, 0)
```

With a datetime object we can do things like printing the exact day of the week.

[4]:
```
date.strftime('%A')
```

[4]:
```
'Saturday'
```

Now we note that when it comes to representing dates as a time series we need to represent these series of dates in an array format. This is where we use the **Numpy** date series datatype called **datetime64**.

```
[6]: date = np.array('2017-07-04', dtype=np.datetime64)
     date
```

```
[6]: array('2017-07-04', dtype='datetime64[D]')
```

Once our date has been initialized and formatted in a datetime64 type, we can carry out vectorized operations on it.

```
[7]: date + np.arange(12)
```

```
[7]: array(['2017-07-04', '2017-07-05', '2017-07-06', '2017-07-07',
            '2017-07-08', '2017-07-09', '2017-07-10', '2017-07-11',
            '2017-07-12', '2017-07-13', '2017-07-14', '2017-07-15'],
           dtype='datetime64[D]')
```

What significantly improved computation with time series data was the Pandas **timestamp** object which combines the interesting functionalities of datetime, dateutil and the vectorized array computation framework of datetime64.

```
[8]: date = pd.to_datetime('4th of July, 2015')
     date
```

```
[8]: Timestamp('2015-07-04 00:00:00')
```

```
[31]: date.strftime('%A')
```

```
[31]: 'Saturday'
```

```
[32]: date + pd.to_timedelta(np.arange(12), 'D')
```

```
[32]: DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
                     '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
                     '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
                    dtype='datetime64[ns]', freq=None)
```

Things start to get interesting when are able to index time series data using the **DatetimeIndex** module. We are essentially indexing our data here by date.

```
[14]: index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                                '2015-07-04', '2015-08-04'])
      data = pd.Series([1,2,3,4], index=index)
      data
```

```
[14]: 2014-07-04    1
      2014-08-04    2
```

```
2015-07-04    3
2015-08-04    4
dtype: int64
```

Now the standard pandas series indexing functionality can be used by indexing with dates.

```
[16]: data['2014-07-04':'2014-08-04']
```

```
[16]: 2014-07-04    1
      2014-08-04    2
      dtype: int64
```

```
[17]: data['2015']
```

```
[17]: 2015-07-04    3
      2015-08-04    4
      dtype: int64
```

We note that the **to_datetime** function can be used to parse a variety of loosely defined date formats to a strict timestamp object.

```
[19]: dates = pd.to_datetime([datetime(2015,7,3), '4th of July,2015',
                              '2015-Jul-6', '07-07-2015', '20150708'])
      dates
```

```
[19]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                     '2015-07-08'],
                    dtype='datetime64[ns]', freq=None)
```

The **DatetimeIndex** is essentially converted to a **periodIndex** and note that 'D' indicates daily frequency. Converting to this format would give us a time period in days.

```
[20]: dates.to_period('D')
```

```
[20]: PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                   '2015-07-08'],
                  dtype='period[D]', freq='D')
```

```
[21]: dates - dates[0]
```

```
[21]: TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
      dtype='timedelta64[ns]', freq=None)
```

Now in order to conveniently create date sequences or time series, we use functions like **date_range** for timestamps, **period_range** for periods and **timedelta_range** for time deltas. We typically put in a start date and end date, sometimes along with a step value.

```
[23]: pd.date_range('2015-07-03', '2015-07-10')
```

```
[23]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                     '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                    dtype='datetime64[ns]', freq='D')
```

```
[24]: pd.date_range('2015-07-03', periods=8)
```

```
[24]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                     '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                    dtype='datetime64[ns]', freq='D')
```

We can create monthly date periods as well by specifying 'M' in the **period_range** function.

```
[26]: pd.period_range('2015-8', periods=8, freq='M')
```

```
[26]: PeriodIndex(['2015-08', '2015-09', '2015-10', '2015-11', '2015-12', '2016-01',
                   '2016-02', '2016-03'],
                  dtype='period[M]', freq='M')
```

Now to note some important features and symbols. In time series analysis we often use different frequencies of time data - years, months, quarters, days. Following are symbols that need to be specified to provide the desired frequency spacing in our data:

1. **D** - calendar day
2. **B** - business day
3. **M** - month end
4. **BM** - business month end
5. **Q** - quarter end
6. **W** - weekly
7. **BQ** - business quarter end
8. **A** - year end
9. **BA** - business year end
10. **H** - hours
11. **BH** - business hours

We further note that adding a suffix **S** to these symbols make our frequencies count from the start of the frequency period.

1. **MS** - month start
2. **BMS** - business month start

Additionally we can even specify the particular month or year from which our frequency periods start by specifying the month suffix - **Q-JAN, BQ-FEB, A-JAN**. Now all this allows us to create Pandas time series offsets which are enabled using **pd.tseries.offsets**. Here we show how to create offsets or time frequencies using business days.

```
[34]: from pandas.tseries.offsets import BDay
      dates = pd.date_range('2015-07-04', periods=10, freq=BDay())
      dates
```

```
[34]: DatetimeIndex(['2015-07-06', '2015-07-07', '2015-07-08', '2015-07-09',
                     '2015-07-10', '2015-07-13', '2015-07-14', '2015-07-15',
                     '2015-07-16', '2015-07-17'],
                    dtype='datetime64[ns]', freq='B')
```

An important functionality of all these time series modules is that they allow us to use dates and times as indices to our data and hence we are able to access data pertaining to time periods with relative ease.

Note that for subsequent sections you will have to install the module **datareader** - which essentially downloads financial data - by using the install function - **conda install pandas-datareader**. Note that this install function is for Python running on Anaconda. If your python is running on some other IDE then you might want to use **pip install pandas-datareader** or refer to documentation on the net.

```
[43]: from pandas_datareader import data

      goog = data.DataReader('GOOG', start='2004', end='2016',
                             data_source='yahoo')
      goog.head()
```

```
[43]:                  High        Low       Open      Close      Volume  Adj Close
      Date
      2004-08-19  51.835709  47.800831  49.813286  49.982655  44871300.0  49.982655
      2004-08-20  54.336334  50.062355  50.316402  53.952770  22942800.0  53.952770
      2004-08-23  56.528118  54.321388  55.168217  54.495735  18342800.0  54.495735
      2004-08-24  55.591629  51.591621  55.412300  52.239193  15319700.0  52.239193
      2004-08-25  53.798351  51.746044  52.284027  52.802086   9232100.0  52.802086
```

We can now observe the trajectory of Google’s stock price over this time period by using the **close** price as our plotting measure.

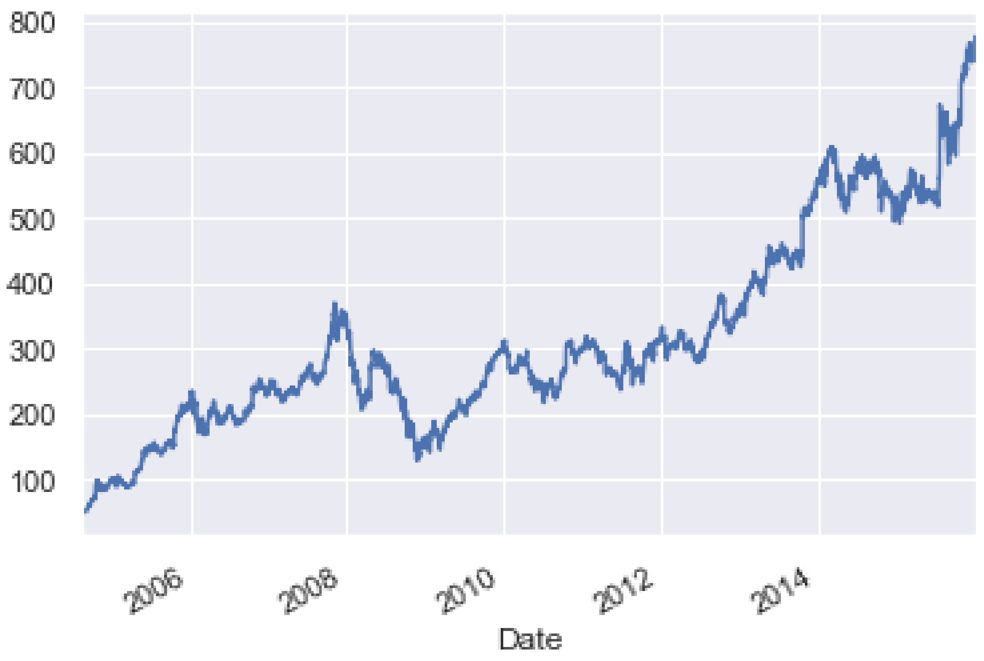
```
[44]: goog = goog['Close']
```

```
[44]: Date
      2004-08-19     49.982655
      2004-08-20     53.952770
      2004-08-23     54.495735
      2004-08-24     52.239193
      2004-08-25     52.802086
                       …
      2015-12-24    748.400024
      2015-12-28    762.510010
      2015-12-29    776.599976
      2015-12-30    771.000000
      2015-12-31    758.880005
      Name: Close, Length: 2863, dtype: float64
```

```
[45]: %matplotlib inline
      import matplotlib.pyplot as plt
      import seaborn
      seaborn.set()

      goog.plot()
```

```
[45]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1cd50ad0>
```

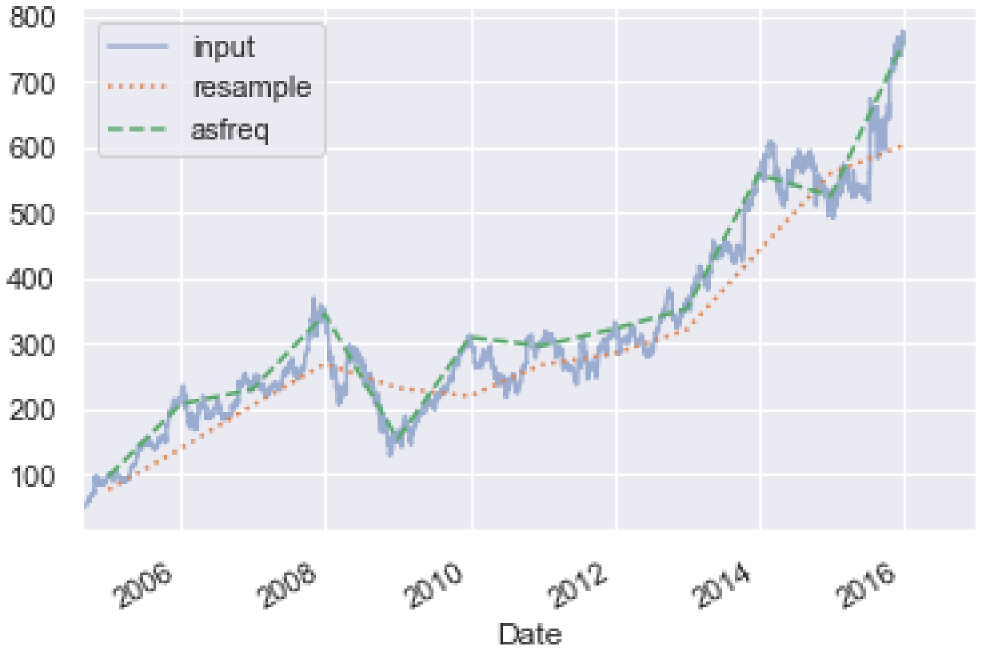


Going more indepth into this type of analysis, we can also **resample** the data at a higher or lower frequency. Note that **resampling** refers to plotting the data as per a different frequency metric. The two function to be used in this regard are - **resample()** which does *data aggregation* with successive time value and **asfreq()** which merely selects the data values at specific time points. Below we can present all these approaches as re resample the Google stock price data as per **Business year**. Note that we are decreasing the frequency measure to years hence we are essentially **down sampling** the data.

1. At each time point, **resample** plots the average of the previous year.
2. At each time point, **asfreq** reports the value of the stock at the end of the year.

```
[48]: goog.plot(alpha=0.5, style='-')
      goog.resample('BA').mean().plot(style=':')
      goog.asfreq('BA').plot(style='--')
      plt.legend(['input', 'resample', 'asfreq'], loc='upper left')
```

```
[48]: <matplotlib.legend.Legend at 0x1a1d06c690>
```



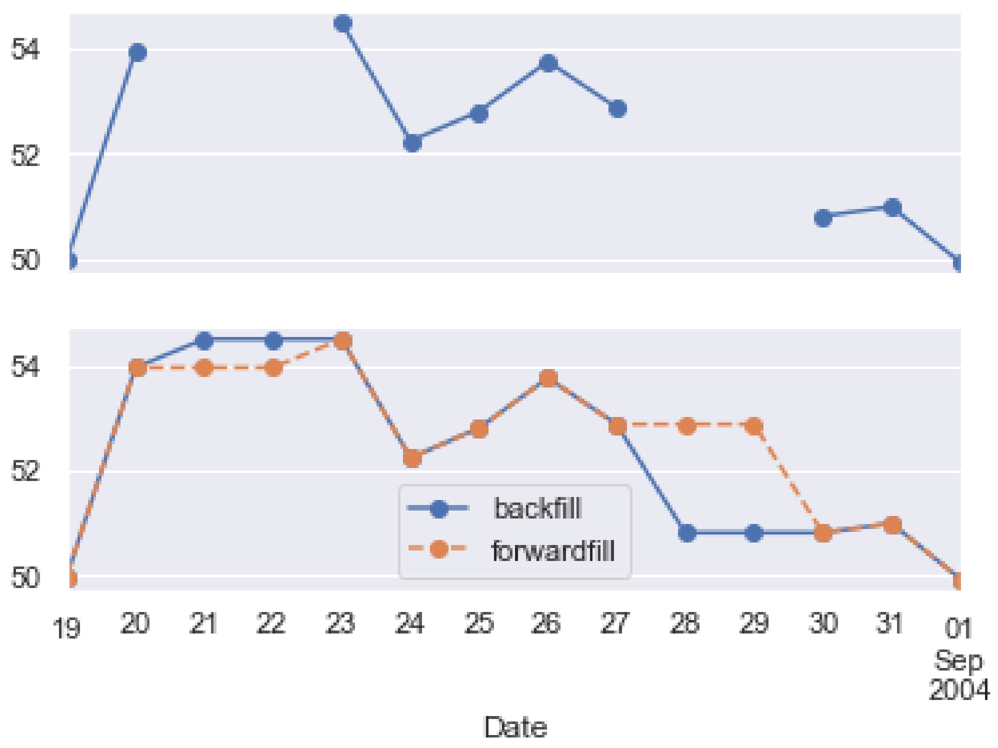
Now we will demonstrate **up sampling** using daily frequency. Note that in daily frequency we might encounter **NA** values because of weekends and such. We will specify exactly how these are represented.

[51]:
```
fig, ax = plt.subplots(2, sharex=True)
data = goog.iloc[:10]

data.asfreq('D').plot(ax=ax[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=ax[1], style='-o')
data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')
ax[1].legend(['backfill', 'forwardfill'])
```

[51]:
```
<matplotlib.legend.Legend at 0x1a1d6034d0>
```



References - Python DataScience handbook