How to use dates & times with pandas

MANIPULATING TIME SERIES DATA IN PYTHON

Stefan Jansen

Founder & Lead Data Scientist at Applied Artificial Intelligence

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
```

Timestamp is the pandas equivalent of python's Datetime and is interchangeable with it in most cases.

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

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

If you display the timestamp, you'll notice that the time has been automatically set to midnight.

```
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.weekday_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

It has a method to convert between frequencies.

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

More building blocks: pd.Period & freq

```python
period + 2
```

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

```python
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` The default is daily frequency.

```
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

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

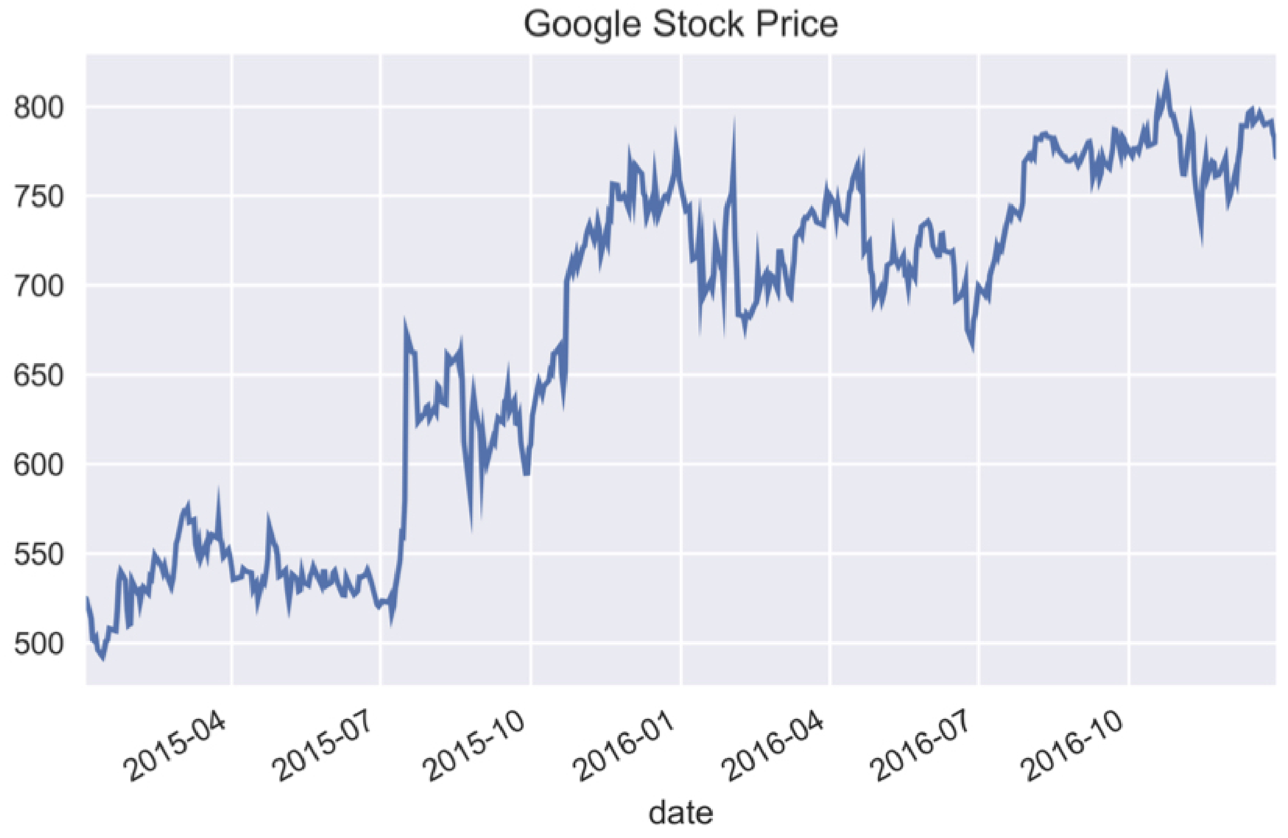
```
google.set_index('date', inplace=True)
google.info()
```

```
<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()
```

With a DateTime Index, pandas automatically creates reasonably spaced date labels for the x axis.



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
```

includes the end date

```
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()
```

As a result, the DateTimeIndex now contains many dates where stock wasn't bought or sold.

```
             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

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='
```

```
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

```python
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