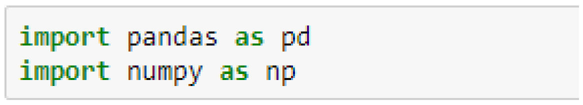
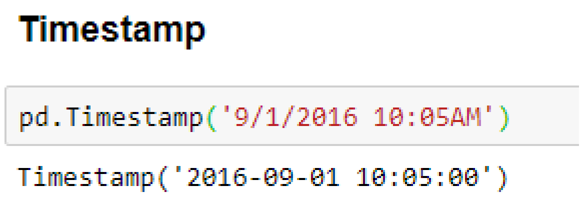
Date Functionality

Now for today's tutorial, we'll be looking at the time series and date functionally in pandas.
Pandas has four main time related classes.
Timestamp, DatetimeIndex, Period, and PeriodIndex.
First, let's look at Timestamp.
Timestamp represents a single timestamp and associates values with points in time.
For example, let's create a timestamp using a string 9/1/2016 10:05AM, and here we have our timestamp.
Timestamp is interchangeable with Python's datetime in most cases.



```
import pandas as pd
import numpy as np
```

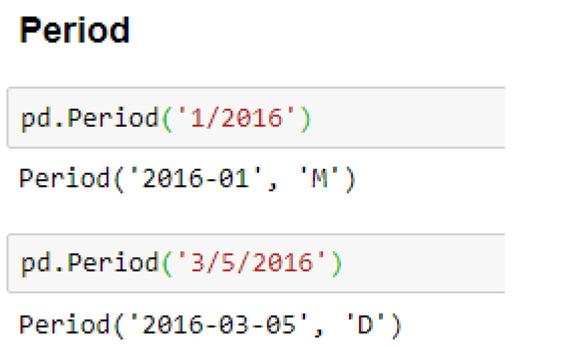


Timestamp

```
pd.Timestamp('9/1/2016 10:05AM')
```

```
Timestamp('2016-09-01 10:05:00')
```

Suppose we weren't interested in a specific point in time, and instead wanted a span of time.
This is where Period comes into play.
Period represents a single time span, such as a specific day or month.
Here we are creating a period that is January 2016, and here's an example of a period that is March 5th, 2016.



Period

```
pd.Period('1/2016')
```

```
Period('2016-01', 'M')
```

```
pd.Period('3/5/2016')
```

```
Period('2016-03-05', 'D')
```

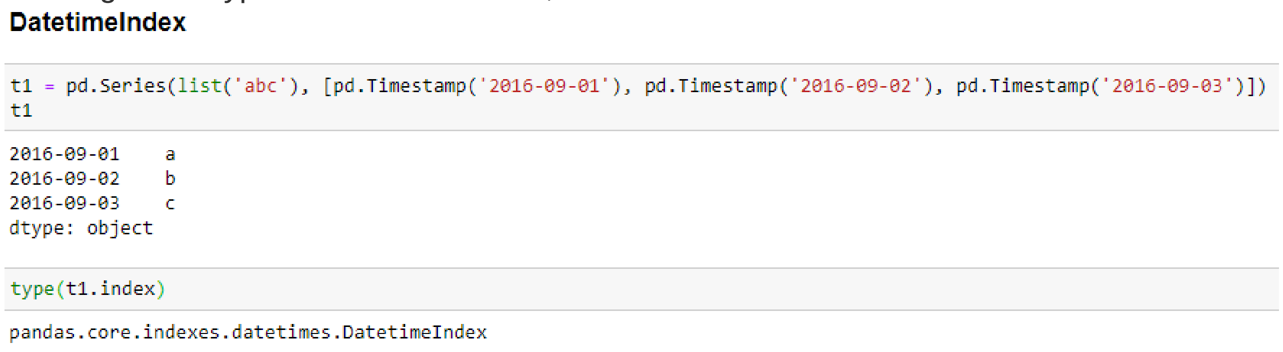
The index of a timestamp is DatetimeIndex.

Let's look at a quick example.

First, let's create our example series t1, we'll use the Timestamp of September 1st, 2nd and 3rd of 2016.

When we look at the series, each Timestamp is the index and has a value associated with it, in this case, a, b and c.

Looking at the type of our series index, we see that it's DatetimeIndex.



DatetimeIndex

```
t1 = pd.Series(list('abc'), [pd.Timestamp('2016-09-01'), pd.Timestamp('2016-09-02'), pd.Timestamp('2016-09-03')])
t1
```

```
2016-09-01    a
2016-09-02    b
2016-09-03    c
dtype: object
```

```
type(t1.index)
```

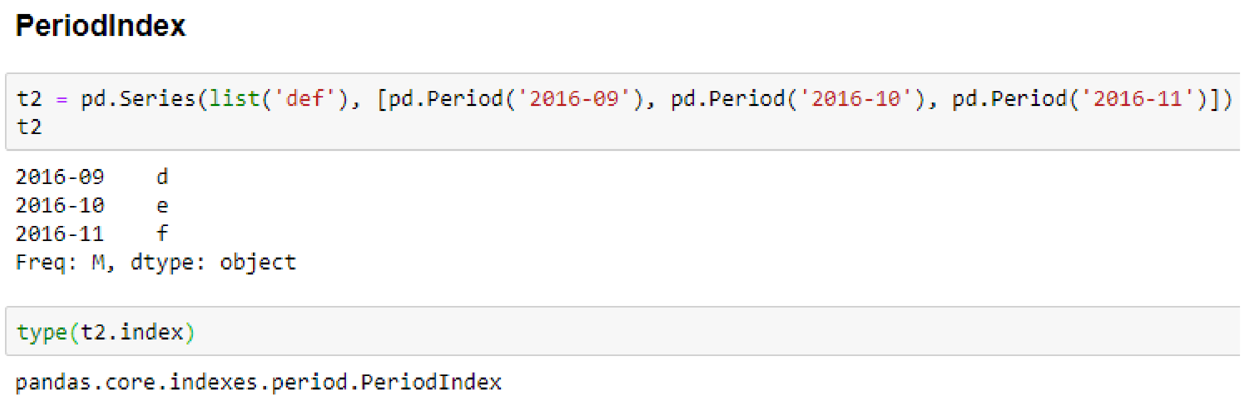
```
pandas.core.indexes.datetimes.DatetimeIndex
```

Similarly, the index of period is PeriodIndex.

Let's create another example series t2.

This time, we'll use the values d, e, and f and match them with the period September, October and November 2016.

Looking at the type of the ts2.index, we can see that it's PeriodIndex.



PeriodIndex

```
t2 = pd.Series(list('def'), [pd.Period('2016-09'), pd.Period('2016-10'), pd.Period('2016-11')])
t2
```

```
2016-09    d
2016-10    e
2016-11    f
Freq: M, dtype: object
```

```
type(t2.index)
```

```
pandas.core.indexes.period.PeriodIndex
```

Now, let's look into how to convert to Datetime.

Suppose we have a list of dates as strings. If we create a DataFrame using these dates as the index. And some randomly generated data; this is the DataFrame we get.

Converting to Datetime

```
d1 = ['2 June 2013', 'Aug 29, 2014', '2015-06-26', '7/12/16']
ts3 = pd.DataFrame(np.random.randint(10, 100, (4,2)), index=d1, columns=list('ab'))
ts3
```

	a	b
2 June 2013	56	54
Aug 29, 2014	91	16
2015-06-26	93	61
7/12/16	25	57

```
ts3.index = pd.to_datetime(ts3.index)
ts3
```

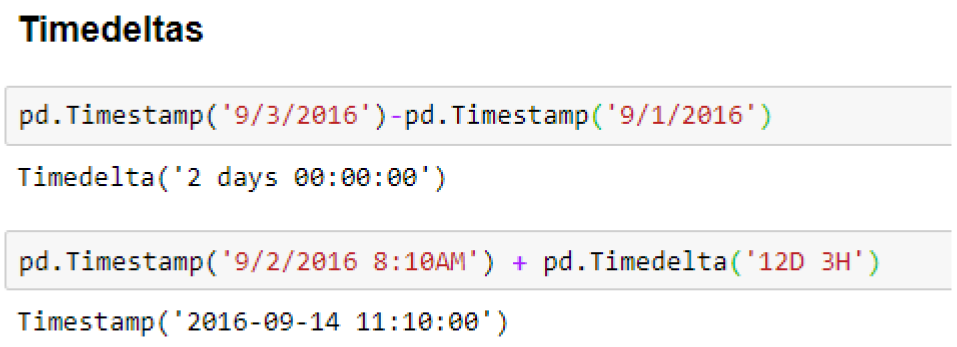
	a	b
2013-06-02	56	54
2014-08-29	91	16
2015-06-26	93	61
2016-07-12	25	57

Looking at the index we can see that it’s pretty messy and the dates are all in different formats.
Using pandas *to_datetime*, pandas will try to convert these to Datetime and put them in a standard format.
to_datetime also has options to change the date parse order.
For example, we can pass in the argument *dayfirst* = True to parse the date in European date format.

```
pd.to_datetime('4.7.12', dayfirst=True)
```

```
Timestamp('2012-07-04 00:00:00')
```

Timedeltas are differences in times. We can see that when we take the difference between September 3rd and September 1st, we get a Timedelta of two days.
We can also do something like find what the date and time is for 12 days and three hours past September 2nd, at 8:10 AM.



Timedeltas

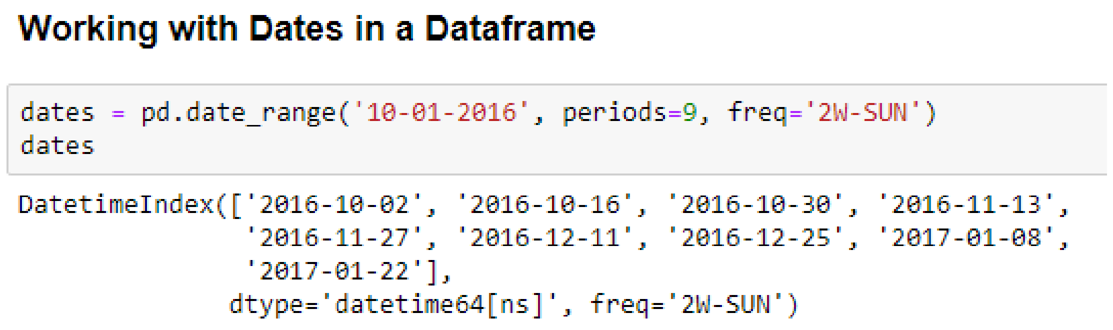
```
pd.Timestamp('9/3/2016')-pd.Timestamp('9/1/2016')
```

```
Timedelta('2 days 00:00:00')
```

```
pd.Timestamp('9/2/2016 8:10AM') + pd.Timedelta('12D 3H')
```

```
Timestamp('2016-09-14 11:10:00')
```

Next, let's look at a few tricks for working with dates in a DataFrame.
Suppose we want to look at nine measurements, taken bi-weekly, every Sunday, starting in October 2016.
Using date_range, we can create this DatetimeIndex.



Working with Dates in a Dataframe

```
dates = pd.date_range('10-01-2016', periods=9, freq='2W-SUN')
dates
```

```
DatetimeIndex(['2016-10-02', '2016-10-16', '2016-10-30', '2016-11-13',
               '2016-11-27', '2016-12-11', '2016-12-25', '2017-01-08',
               '2017-01-22'],
              dtype='datetime64[ns]', freq='2W-SUN')
```

Now, let's create a DataFrame using these dates, and some random data, and see what we can do with it.

```
df = pd.DataFrame({'Count 1': 100 + np.random.randint(-5, 10, 9).cumsum(),
                  'Count 2': 120 + np.random.randint(-5, 10, 9)}, index=dates)
df
```

	Count 1	Count 2
2016-10-02	95	116
2016-10-16	99	128
2016-10-30	103	119
2016-11-13	99	126
2016-11-27	97	121
2016-12-11	99	122
2016-12-25	107	117
2017-01-08	103	119
2017-01-22	100	119

numpy.cumsum() function is used when we want to compute the cumulative sum of array elements over a given axis.

First, we can check what day of the week a specific date is.

For example, here we can see that all the dates in our index are on a Sunday.

```
df.index.weekday_name
```

```
Index(['Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday',
       'Sunday', 'Sunday'],
      dtype='object')
```

We can use diff to find the difference between each date's values.

```
df.diff()
```

	Count 1	Count 2
2016-10-02	NaN	NaN
2016-10-16	4.0	12.0
2016-10-30	4.0	-9.0
2016-11-13	-4.0	7.0
2016-11-27	-2.0	-5.0
2016-12-11	2.0	1.0
2016-12-25	8.0	-5.0
2017-01-08	-4.0	2.0
2017-01-22	-3.0	0.0

Suppose we wanted to know what the mean count is for each month in our DataFrame. We can do this using resample.

```
df.resample('M').mean()
```

	Count 1	Count 2
2016-10-31	99.0	121.0
2016-11-30	98.0	123.5
2016-12-31	103.0	119.5
2017-01-31	101.5	119.0

We can use partial string indexing to find values from a particular year, or from a particular month, or we can even slice on a range of dates.
For example, here we only want the values from December 2016 onwards.

```
df['2017']
```

	Count 1	Count 2
2017-01-08	103	119
2017-01-22	100	119

```
df['2016-12']
```

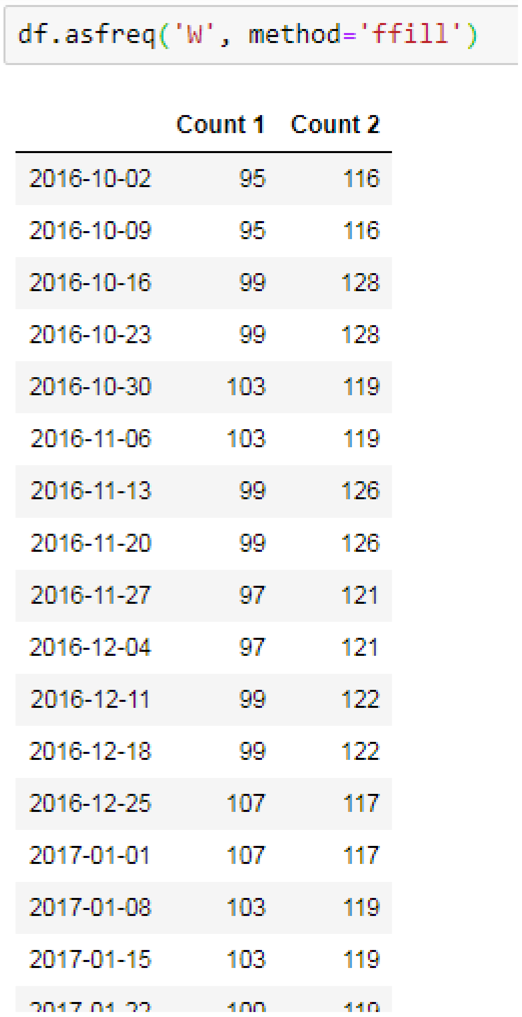
	Count 1	Count 2
2016-12-11	99	122
2016-12-25	107	117

```
df['2016-12':]
```

	Count 1	Count 2
2016-12-11	99	122
2016-12-25	107	117
2017-01-08	103	119
2017-01-22	100	119

Another cool thing we can do is change the frequency of our dates in our DataFrame using *asfreq*.
If we use this to change the frequency from bi-weekly to weekly, we'll end up with missing values every other week.

So let's use the forward fill method on those missing values.



```
df.asfreq('W', method='ffill')
```

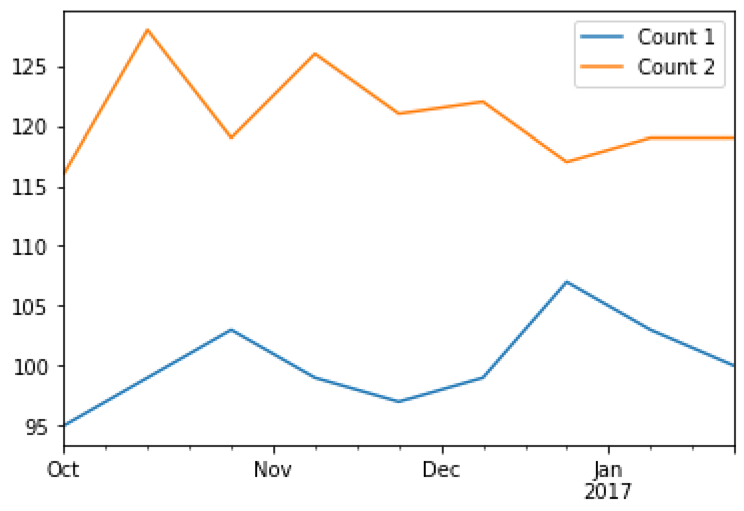
	Count 1	Count 2
2016-10-02	95	116
2016-10-09	95	116
2016-10-16	99	128
2016-10-23	99	128
2016-10-30	103	119
2016-11-06	103	119
2016-11-13	99	126
2016-11-20	99	126
2016-11-27	97	121
2016-12-04	97	121
2016-12-11	99	122
2016-12-18	99	122
2016-12-25	107	117
2017-01-01	107	117
2017-01-08	103	119
2017-01-15	103	119
2017-01-22	[illegible]	[illegible]

One last thing I wanted to briefly touch upon is plotting time series. Importing matplotlib.pyplot, and using the iPython magic %mapplotlib inline, will allow you to visualize the time series in the notebook.

```
import matplotlib.pyplot as plt
%matplotlib inline

df.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x58ead48>
```



In the next course, we will learn more about understanding and creating visualizations.