Time Series

July 19, 2015

1 Time series

From *Python for Data Analysis*:

> Time series data is an important form of structured data in many different dielfds, such as finance, economics, ecology, neuroscience, and physics. Anything that is observed or measured at many points in time forms a time series. Many time series are *fixed frequency*, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be *irregular* without a fixed unit or time or offset between units. How you mark and refer to time series data depends on the application and you may have one of the following:
>
> - *timestamps*, specific instants in time
> - *fixed periods*, such as the month January 2007 or the full year 2010
> - *intervals* of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals
> - Experiment or elapsed time; each timestamp is a measure of time relative to a particular start time. For example, the diameter of a cookie baking each second since being placed in the oven
>
> `Pandas` provides a standard set of time series tools and data algorithms. With this you can efficiently work with very large time series and easily slice and dice, aggregate, and resample irregular and fixed frequency time series. As you might guess, many of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.

```
In [1]: from __future__ import division
        from pandas import Series, DataFrame
        import pandas as pd
        from numpy.random import randn
        import numpy as np
        pd.options.display.max_rows = 12
        np.set_printoptions(precision=4, suppress=True)
        import matplotlib.pyplot as plt
        plt.rc('figure', figsize=(12, 4))
```

```
In [2]: %matplotlib inline
```

1.1 Date and Time Data Types and Tools

In general, dealing with date arithmetic is *hard*. Luckily, `Python` has a robust library that implements `datetime` objects, which handle all of the annoying bits of date manipulation in a powerful way.

```
In [3]: from datetime import datetime
        now = datetime.now()
        now
```

```
Out[3]: datetime.datetime(2015, 7, 19, 18, 9, 34, 631424)
```

Every `datetime` object has a `year`, `month`, and `day` field.

```
In [4]: now.year, now.month, now.day
```

```
Out[4]: (2015, 7, 19)
```

You can do arithmetic on `datetime` objects, which produce `timedelta` objects.

```
In [5]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)
        delta
```

```
Out[5]: datetime.timedelta(926, 56700)
```

`timedelta` objects are very similar to `datetime` objects, with similar fields:

```
In [6]: delta.days
```

```
Out[6]: 926
```

```
In [7]: delta.seconds
```

```
Out[7]: 56700
```

As you expect, arithmetic between `datetime` and `timedelta` objects produce `datetime` objects.

```
In [8]: from datetime import timedelta
        start = datetime(2011, 1, 7)
        start + timedelta(12)
```

```
Out[8]: datetime.datetime(2011, 1, 19, 0, 0)
```

```
In [9]: start - 2 * timedelta(12)
```

```
Out[9]: datetime.datetime(2010, 12, 14, 0, 0)
```

1.1.1 Converting between string and datetime

In general, it is easier to format a string from a `datetime` object than to parse a string date into a `datetime` object.

```
In [10]: stamp = datetime(2011, 1, 3)
```

```
In [11]: str(stamp)
```

```
Out[11]: '2011-01-03 00:00:00'
```

To format a string from a `datetime` object, use the `strftime` method. You can use the standard string-formatting delimiters that are used in computing.

```
In [12]: stamp.strftime('%Y-%m-%d')
```

```
Out[12]: '2011-01-03'
```

To parse a string into a `datetime` object, you can use the `strptime` method, along with the relevant format.

```
In [13]: value = '2011-01-03'
         datetime.strptime(value, '%Y-%m-%d')

Out[13]: datetime.datetime(2011, 1, 3, 0, 0)
```

Of course, this being `Python`, we can easily abstract this process to list form using comprehensions.

```
In [14]: datestrs = ['7/6/2011', '8/6/2011']
         [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]

Out[14]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]
```

Without question, `datetime.strptime` is the best way to parse a date, especially when you know the format a priori. However, it can be a bit annoying to have to write a format spec each time, especially for common date formats. In this case, you can use the `parser.parse` method in the third party `dateutil` package:

```
In [15]: from dateutil.parser import parse
         parse('2011-01-03')

Out[15]: datetime.datetime(2011, 1, 3, 0, 0)
```

`dateutil` is capable of parsing almost any human-intelligible date representation:

```
In [16]: parse('Jan 31, 1997 10:45 PM')

Out[16]: datetime.datetime(1997, 1, 31, 22, 45)
```

In international locales, day appearing before month is very common, so you can pass `dayfirst=True` to indicate this:

```
In [17]: parse('6/12/2011', dayfirst=True)

Out[17]: datetime.datetime(2011, 12, 6, 0, 0)
```

`Pandas` is generally oriented toward working with arrays of dates, whether used as an index or a column in a `DataFrame`. The `to_datetime` method parses many different kinds of date representations. Standard date formats like ISO8601 can be parsed very quickly.

```
In [18]: datestrs

Out[18]: ['7/6/2011', '8/6/2011']

In [19]: pd.to_datetime(datestrs)

Out[19]: DatetimeIndex(['2011-07-06', '2011-08-06'], dtype='datetime64[ns]', freq=None, tz=None)
```

Notice that the `Pandas` object at work behind the scenes here is the `DatetimeIndex`, which is a subclass of `Index`. More on this later. `to_datetime` also handles values that should be considered missing (`None`, empty string, etc.):

```
In [20]: idx = pd.to_datetime(datestrs + [None])
         idx

Out[20]: DatetimeIndex(['2011-07-06', '2011-08-06', 'NaT'], dtype='datetime64[ns]', freq=None, tz=None)

In [21]: idx[2]

Out[21]: NaT

In [22]: pd.isnull(idx)

Out[22]: array([False, False,  True], dtype=bool)
```

`datetime` objects also have a number of locale-specific formatting options for systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems.

1.2 Time Series Basics

The most basic kind of time series object in `Pandas` is a `Series` indexed by timestamps, which is often represented external to `Pandas` as `Python` strings or `datetime` objects.

```
In [23]: from datetime import datetime
         dates = [datetime(2011, 1, 2), datetime(2011, 1, 5), datetime(2011, 1, 7),
                  datetime(2011, 1, 8), datetime(2011, 1, 10), datetime(2011, 1, 12)]
         ts = Series(np.random.randn(6), index=dates)
         ts

Out[23]: 2011-01-02    1.468516
         2011-01-05   -2.225967
         2011-01-07   -0.661142
         2011-01-08    0.897391
         2011-01-10   -1.298501
         2011-01-12    0.023799
         dtype: float64
```

Under the hood, these `datetime` objects have been put in a `DatetimeIndex`, and the variable `ts` is now of type `TimeSeries`.

```
In [24]: type(ts)
         # note: output changed to "pandas.core.series.Series"

Out[24]: pandas.core.series.Series

In [25]: ts.index

Out[25]: DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
                        '2011-01-10', '2011-01-12'],
                       dtype='datetime64[ns]', freq=None, tz=None)
```

Like other `Series`, arithmetic operations between differently-indexed time series automatically align on the dates:

```
In [26]: ts + ts[::2]

Out[26]: 2011-01-02    2.937031
         2011-01-05         NaN
         2011-01-07   -1.322284
         2011-01-08         NaN
         2011-01-10   -2.597002
         2011-01-12         NaN
         dtype: float64
```

`Pandas` stores timestamps using `NumPy`'s `datetime64` date type at the nanosecond resolution:

```
In [27]: ts.index.dtype
         # note: output changed from dtype('datetime64[ns]') to dtype('<M8[ns]')

Out[27]: dtype('<M8[ns]')
```

Scalar values from a `DatetimeIndex` are `Pandas` `Timestamp` objects

```
In [28]: stamp = ts.index[0]
         stamp
         # note: output changed from <Timestamp: 2011-01-02 00:00:00> to Timestamp('2011-01-02 00:00:00

Out[28]: Timestamp('2011-01-02 00:00:00')
```

A `Timestamp` can be substituted anywhere you would use a `datetime` object. Additionally, it can store frequency information (if any) and understands how to do time zone conversions and other kinds of manipulations. More on both of these things later.

1.2.1 Indexing, selection, subsetting

`TimeSeries` is a subclass of `Series` and thus behaves in the same way with regard to indexing and selecting data based on label:

```
In [29]: stamp = ts.index[2]
         ts[stamp]

Out[29]: -0.66114218523925783
```

As a convenience, you can also pass a string that is interpretable as a date:

```
In [30]: ts['1/10/2011']

Out[30]: -1.2985009506418492

In [31]: ts['20110110']

Out[31]: -1.2985009506418492
```

For longer time series, a year or only a year and month can be passed to easily select slices of data:

```
In [32]: longer_ts = Series(np.random.randn(1000),
                            index=pd.date_range('1/1/2000', periods=1000))
         longer_ts

Out[32]: 2000-01-01    0.136919
         2000-01-02   -0.758297
         2000-01-03    0.525571
         2000-01-04   -0.208681
         2000-01-05    0.490768
         2000-01-06   -0.379107
                         ...
         2002-09-21   -0.381494
         2002-09-22   -0.454672
         2002-09-23   -1.129404
         2002-09-24    0.179486
         2002-09-25    0.146503
         2002-09-26    0.347724
         Freq: D, dtype: float64

In [33]: longer_ts['2001']

Out[33]: 2001-01-01   -1.577440
         2001-01-02   -0.561100
         2001-01-03    0.202502
         2001-01-04   -0.394490
         2001-01-05    1.778403
         2001-01-06    0.563747
                         ...
         2001-12-26   -1.849043
         2001-12-27    0.346492
         2001-12-28   -1.586010
         2001-12-29   -0.823232
         2001-12-30    0.205626
         2001-12-31    0.771505
         Freq: D, dtype: float64
```

```
In [34]: longer_ts['2001-05']

Out[34]: 2001-05-01    0.417848
         2001-05-02   -0.263080
         2001-05-03   -0.644724
         2001-05-04   -1.393179
         2001-05-05   -0.643053
         2001-05-06   -0.572408
                         ...
         2001-05-26   -0.671731
         2001-05-27   -0.504401
         2001-05-28    1.842141
         2001-05-29   -0.606931
         2001-05-30   -1.125938
         2001-05-31   -0.642261
         Freq: D, dtype: float64
```

Slicing with dates works just like with a regular `Series`

```
In [35]: ts[datetime(2011, 1, 7):]

Out[35]: 2011-01-07   -0.661142
         2011-01-08    0.897391
         2011-01-10   -1.298501
         2011-01-12    0.023799
         dtype: float64
```

Because most time series data is ordered chronologically, you can slice with timestamps not contained in a time series to perform a range query:

```
In [36]: ts

Out[36]: 2011-01-02    1.468516
         2011-01-05   -2.225967
         2011-01-07   -0.661142
         2011-01-08    0.897391
         2011-01-10   -1.298501
         2011-01-12    0.023799
         dtype: float64

In [37]: ts['1/6/2011':'1/11/2011']

Out[37]: 2011-01-07   -0.661142
         2011-01-08    0.897391
         2011-01-10   -1.298501
         dtype: float64
```

As before you can pass either a string date, `datetime`, or `Timestamp`. Remember that slicing in this manner produces views on the source time series just like slicing `NumPy` arrays. There is an equivalent instance method `truncate` which slices a `TimeSeries` between two dates:

```
In [38]: ts.truncate(after='1/9/2011')

Out[38]: 2011-01-02    1.468516
         2011-01-05   -2.225967
         2011-01-07   -0.661142
         2011-01-08    0.897391
         dtype: float64
```

All of the above holds true for DataFrame as well, indexing on its rows:

```
In [39]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')
         long_df = DataFrame(np.random.randn(100, 4),
                             index=dates,
                             columns=['Colorado', 'Texas', 'New York', 'Ohio'])
         long_df.ix['5-2001']

Out[39]:             Colorado     Texas  New York      Ohio
         2001-05-02  0.017935  0.661052  1.790387  1.207539
         2001-05-09 -1.531336 -1.347647 -2.066621  0.703189
         2001-05-16  0.115050 -0.090165 -2.142895 -2.192053
         2001-05-23  0.970376 -1.885132  0.964201 -0.558697
         2001-05-30 -0.711281 -0.687192 -1.189195  0.449382
```

1.2.2 Time series with duplicate indices

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

```
In [40]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',
                                   '1/3/2000'])
         dup_ts = Series(np.arange(5), index=dates)
         dup_ts

Out[40]: 2000-01-01    0
         2000-01-02    1
         2000-01-02    2
         2000-01-02    3
         2000-01-03    4
         dtype: int64
```

We can tell that the index is not unique by checking its is_unique property:

```
In [41]: dup_ts.index.is_unique

Out[41]: False
```

Indexing into this time series will now either produce scalar values or slices depending on whether a timestamp is duplicated:

```
In [42]: dup_ts['1/3/2000']  # not duplicated

Out[42]: 4

In [43]: dup_ts['1/2/2000']  # duplicated

Out[43]: 2000-01-02    1
         2000-01-02    2
         2000-01-02    3
         dtype: int64
```

Suppose you want to aggregate the data having non-unique timestamps. One way to do this is to use groupby and pass level=0 (the only level of indexing!):

```
In [44]: grouped = dup_ts.groupby(level=0)
         grouped.mean()
```

```
Out[44]: 2000-01-01    0
         2000-01-02    2
         2000-01-03    4
         dtype: int64
```

```
In [45]: grouped.count()
```

```
Out[45]: 2000-01-01    1
         2000-01-02    3
         2000-01-03    1
         dtype: int64
```

1.3 Date ranges, Frequencies, and Shifting

Generic time series in `Pandas` are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it's often desirable to work relative to a fixed frequency, such as daily, monthly, or even 15 minutes, even if that means introducing missing values into a time series. Fortunately `Pandas` has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed frequency date ranges. For example, in the example time series, converting it to be fixed daily frequency can be accomplished by calling `resample`:

```
In [46]: ts
```

```
Out[46]: 2011-01-02    1.468516
         2011-01-05   -2.225967
         2011-01-07   -0.661142
         2011-01-08    0.897391
         2011-01-10   -1.298501
         2011-01-12    0.023799
         dtype: float64
```

```
In [47]: ts.resample('D')
```

```
Out[47]: 2011-01-02    1.468516
         2011-01-03         NaN
         2011-01-04         NaN
         2011-01-05   -2.225967
         2011-01-06         NaN
         2011-01-07   -0.661142
         2011-01-08    0.897391
         2011-01-09         NaN
         2011-01-10   -1.298501
         2011-01-11         NaN
         2011-01-12    0.023799
         Freq: D, dtype: float64
```

Conversion between frequencies or *resampling* is a big enough topic to have its own section later. Here, we'll see how to use the base frequencies and multiples thereof.

1.3.1 Generating date ranges

You may have guessed that `pandas.date_range` is responsible for generating a `DatetimeIndex` with an indicated length according to a particular frequency:

```
In [48]: index = pd.date_range('4/1/2012', '6/1/2012')
         index
```

```
Out[48]: DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
                        '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
                        '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
                        '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
                        '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
                        '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
                        '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
                        '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
                        '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
                        '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
                        '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
                        '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
                        '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
                        '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
                        '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
                        '2012-05-31', '2012-06-01'],
                       dtype='datetime64[ns]', freq='D', tz=None)
```

By default, `date_range` generates daily timestamps. If you pass only a start or end date, you must pass a number of periods to generate:

```
In [49]: pd.date_range(start='4/1/2012', periods=20)

Out[49]: DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
                        '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
                        '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
                        '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
                        '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
                       dtype='datetime64[ns]', freq='D', tz=None)

In [50]: pd.date_range(end='6/1/2012', periods=20)

Out[50]: DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',
                        '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
                        '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
                        '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
                        '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
                       dtype='datetime64[ns]', freq='D', tz=None)
```

The start and end dates define strict boundaries for the generated date index. For example, if you wanted a date index containing the last business day of each month, you would pass the `'BM'` frequency (business end of month) and only dates falling on or inside the date interval will be included:

```
In [51]: pd.date_range('1/1/2000', '12/1/2000', freq='BM')

Out[51]: DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',
                        '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
                        '2000-09-29', '2000-10-31', '2000-11-30'],
                       dtype='datetime64[ns]', freq='BM', tz=None)
```

`date_range` by default preserves the time (if any) or the start or end timestamp:

```
In [52]: pd.date_range('5/2/2012 12:56:31', periods=5)

Out[52]: DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
                        '2012-05-04 12:56:31', '2012-05-05 12:56:31',
                        '2012-05-06 12:56:31'],
                       dtype='datetime64[ns]', freq='D', tz=None)
```

Sometimes you will have start or end dates with time information but want to generate a set of timestamps *normalized* to midnight as a convention. To do this, there is a `normalize` option:

```
In [53]: pd.date_range('5/2/2012 12:56:31', periods=5, normalize=True)

Out[53]: DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',
                        '2012-05-06'],
                       dtype='datetime64[ns]', freq='D', tz=None)
```

1.3.2 Frequencies and Date Offsets

Frequencies in `Pandas` are composed of a *base frequency* and a multiplier. Base frequencies are typically referred to by a string alias, like `'M'` for monthly or `'H'` for hourly. For each base frequency, there is an object defined generally referred to as a *date offset*. For each example, hourly frequency can be represented with the `Hour` class:

```
In [54]: from pandas.tseries.offsets import Hour, Minute
         hour = Hour()
         hour

Out[54]: <Hour>
```

You can define a multiple of an offset by passing an integer:

```
In [55]: four_hours = Hour(4)
         four_hours

Out[55]: <4 * Hours>
```

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like `'H'` or `'4H'`. Putting an integer before the base frequency creates a multiple:

```
In [56]: pd.date_range('1/1/2000', '1/3/2000 23:59', freq='4h')

Out[56]: DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
                        '2000-01-01 08:00:00', '2000-01-01 12:00:00',
                        '2000-01-01 16:00:00', '2000-01-01 20:00:00',
                        '2000-01-02 00:00:00', '2000-01-02 04:00:00',
                        '2000-01-02 08:00:00', '2000-01-02 12:00:00',
                        '2000-01-02 16:00:00', '2000-01-02 20:00:00',
                        '2000-01-03 00:00:00', '2000-01-03 04:00:00',
                        '2000-01-03 08:00:00', '2000-01-03 12:00:00',
                        '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
                       dtype='datetime64[ns]', freq='4H', tz=None)
```

Many offsets can be combined together by addition:

```
In [57]: Hour(2) + Minute(30)

Out[57]: <150 * Minutes>
```

Similarly, you can pass frequency strings like `'2h30min'` which will effectively be parsed to the same expression.

```
In [58]: pd.date_range('1/1/2000', periods=10, freq='1h30min')
```

```
Out[58]: DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
                        '2000-01-01 03:00:00', '2000-01-01 04:30:00',
                        '2000-01-01 06:00:00', '2000-01-01 07:30:00',
                        '2000-01-01 09:00:00', '2000-01-01 10:30:00',
                        '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
                       dtype='datetime64[ns]', freq='90T', tz=None)
```

Some frequencies describe points in time that are not evenly spaced. For example, `'M'` (calendar month end) and `'BM'` (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. For lack of a better term, we will call these *anchored* offsets.

Week of month dates One useful frequency class is "week of month", starting with `WOM`. This enables you to get dates like the third Friday of each month:

```
In [59]: rng = pd.date_range('1/1/2012', '9/1/2012', freq='WOM-3FRI')
         list(rng)

Out[59]: [Timestamp('2012-01-20 00:00:00', offset='WOM-3FRI'),
          Timestamp('2012-02-17 00:00:00', offset='WOM-3FRI'),
          Timestamp('2012-03-16 00:00:00', offset='WOM-3FRI'),
          Timestamp('2012-04-20 00:00:00', offset='WOM-3FRI'),
          Timestamp('2012-05-18 00:00:00', offset='WOM-3FRI'),
          Timestamp('2012-06-15 00:00:00', offset='WOM-3FRI'),
          Timestamp('2012-07-20 00:00:00', offset='WOM-3FRI'),
          Timestamp('2012-08-17 00:00:00', offset='WOM-3FRI')]
```

Traders of US equity options will recognize thse dates as the standard dates of monthly expiry.

1.3.3 Shifting (leading and lagging) data

"Shifting" refers to moving data backward and forward through time. Both `Series` and `DataFrame` have a `shift` method for doing naive shifts forward or backward, leaving the index unmodified:

```
In [60]: ts = Series(np.random.randn(4),
                     index=pd.date_range('1/1/2000', periods=4, freq='M'))
         ts

Out[60]: 2000-01-31   -0.589325
         2000-02-29    0.394207
         2000-03-31    1.465351
         2000-04-30   -0.305100
         Freq: M, dtype: float64

In [61]: ts.shift(2)

Out[61]: 2000-01-31         NaN
         2000-02-29         NaN
         2000-03-31   -0.589325
         2000-04-30    0.394207
         Freq: M, dtype: float64

In [62]: ts.shift(-2)

Out[62]: 2000-01-31    1.465351
         2000-02-29   -0.305100
         2000-03-31         NaN
         2000-04-30         NaN
         Freq: M, dtype: float64
```

A common use of `shift` is computing percent changes in a time series or multiple time series as `DataFrame` columns. This is expressed as ts / ts.shift(1) - 1 Because naive shifts leave the index unmodified, some data is discarded. Thus if the frequency is known, it can be passed to `shift` to advance the timestamps instead of simply the data

```
In [63]: ts.shift(2, freq='M')

Out[63]: 2000-03-31   -0.589325
         2000-04-30    0.394207
         2000-05-31    1.465351
         2000-06-30   -0.305100
         Freq: M, dtype: float64
```

Other frequencies can be passed, too, giving you a lot of flexibility in how to lead and lag the data

```
In [64]: ts.shift(3, freq='D')

Out[64]: 2000-02-03   -0.589325
         2000-03-03    0.394207
         2000-04-03    1.465351
         2000-05-03   -0.305100
         dtype: float64

In [65]: ts.shift(1, freq='3D')

Out[65]: 2000-02-03   -0.589325
         2000-03-03    0.394207
         2000-04-03    1.465351
         2000-05-03   -0.305100
         dtype: float64

In [66]: ts.shift(1, freq='90T')

Out[66]: 2000-01-31 01:30:00   -0.589325
         2000-02-29 01:30:00    0.394207
         2000-03-31 01:30:00    1.465351
         2000-04-30 01:30:00   -0.305100
         dtype: float64
```

Shifting dates with offsets The `Pandas` date offsets can also be used with `datetime` or `Timestamp` objects:

```
In [67]: from pandas.tseries.offsets import Day, MonthEnd
         now = datetime(2011, 11, 17)
         now + 3 * Day()

Out[67]: Timestamp('2011-11-20 00:00:00')
```

If you add an anchored offset like `MonthEnd`, the first increment will `roll forward` a date to the next date according to the frequency rule:

```
In [68]: now + MonthEnd()

Out[68]: Timestamp('2011-11-30 00:00:00')

In [69]: now + MonthEnd(2)

Out[69]: Timestamp('2011-12-31 00:00:00')
```

Anchored offsets can explicitly "roll" dates forward or backward using their `rollforward` and `rollback` methods, respectively:

```
In [70]: offset = MonthEnd()
         offset.rollforward(now)

Out[70]: Timestamp('2011-11-30 00:00:00')

In [71]: offset.rollback(now)

Out[71]: Timestamp('2011-10-31 00:00:00')
```

A clever use of date offsets is to use these methods with groupby:

```
In [72]: ts = Series(np.random.randn(20),
                     index=pd.date_range('1/15/2000', periods=20, freq='4d'))
         ts.groupby(offset.rollforward).mean()

Out[72]: 2000-01-31    0.397220
         2000-02-29    0.303620
         2000-03-31    0.109539
         dtype: float64
```

Of course, an easier and faster way to do this is using `resample` (more on this to come).

```
In [73]: ts.resample('M', how='mean')

Out[73]: 2000-01-31    0.397220
         2000-02-29    0.303620
         2000-03-31    0.109539
         Freq: M, dtype: float64
```

1.4 Time Zone Handling

Working with time zones is a pain. As Americans hold on dearly to daylight savings time, we must pay the price with difficult conversions between time zones. Many time series users choose to work with time series in *coordinated universal time (UTC)* of which time zones can be expressed as offsets.

In `Python` we can use the `pytz` library, based off the Olson *database* of world time zone data.

```
In [74]: import pytz
         pytz.common_timezones[-5:]

Out[74]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

To get a time zone object from `pytz`, use `pytz.timezone`.

```
In [75]: tz = pytz.timezone('US/Eastern')
         tz

Out[75]: <DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>
```

Methods in `Pandas` will accept either time zone names or these objects. Using the names is recommended.

1.4.1 Localization and Conversion

By default, time series in `Pandas` are *time zone naive*. Consider the following time series:

```
In [76]: rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')
         ts = Series(np.random.randn(len(rng)), index=rng)
```

The index's `tz` field is `None`:

```
In [77]: print(ts.index.tz)
```

```
None
```

Date ranges can be generated with a time zone set:

```
In [78]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
```

```
Out[78]: DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
                        '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
                        '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
                        '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',
                        '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],
                       dtype='datetime64[ns]', freq='D', tz='UTC')
```

Conversion from naive to *localized* is handled by the `tz_localize` method

```
In [79]: ts_utc = ts.tz_localize('UTC')
         ts_utc
```

```
Out[79]: 2012-03-09 09:30:00+00:00   -0.401963
         2012-03-10 09:30:00+00:00    1.403076
         2012-03-11 09:30:00+00:00    0.570966
         2012-03-12 09:30:00+00:00   -0.764951
         2012-03-13 09:30:00+00:00   -0.436775
         2012-03-14 09:30:00+00:00   -1.162297
         Freq: D, dtype: float64
```

```
In [80]: ts_utc.index
```

```
Out[80]: DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
                        '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
                        '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00'],
                       dtype='datetime64[ns]', freq='D', tz='UTC')
```

Once a time series has been localized to a particular time zone, it can be converted to another time zone using `tz_convert`.

```
In [81]: ts_utc.tz_convert('US/Eastern')
```

```
Out[81]: 2012-03-09 04:30:00-05:00   -0.401963
         2012-03-10 04:30:00-05:00    1.403076
         2012-03-11 05:30:00-04:00    0.570966
         2012-03-12 05:30:00-04:00   -0.764951
         2012-03-13 05:30:00-04:00   -0.436775
         2012-03-14 05:30:00-04:00   -1.162297
         Freq: D, dtype: float64
```

In this case of the above time series, which straddles a DST transition in the US/Eastern time zone, we could localize to EST and convert to, say, UTC or Berlin time.

```
In [82]: ts_eastern = ts.tz_localize('US/Eastern')
         ts_eastern.tz_convert('UTC')
```

```
Out[82]: 2012-03-09 14:30:00+00:00   -0.401963
         2012-03-10 14:30:00+00:00    1.403076
         2012-03-11 13:30:00+00:00    0.570966
         2012-03-12 13:30:00+00:00   -0.764951
         2012-03-13 13:30:00+00:00   -0.436775
         2012-03-14 13:30:00+00:00   -1.162297
         Freq: D, dtype: float64
```

```
In [83]: ts_eastern.tz_convert('Europe/Berlin')
```

```
Out[83]: 2012-03-09 15:30:00+01:00   -0.401963
         2012-03-10 15:30:00+01:00    1.403076
         2012-03-11 14:30:00+01:00    0.570966
         2012-03-12 14:30:00+01:00   -0.764951
         2012-03-13 14:30:00+01:00   -0.436775
         2012-03-14 14:30:00+01:00   -1.162297
         Freq: D, dtype: float64
```

tz_localize and tz_convert are also instance methods on DatetimeIndex.

```
In [84]: ts.index.tz_localize('Asia/Shanghai')
```

```
Out[84]: DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',
                        '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',
                        '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],
                       dtype='datetime64[ns]', freq='D', tz='Asia/Shanghai')
```

1.4.2 Operations with time zone-aware Timestamp objects

Similar to time series and date ranges, individual Timestamp objects similarly can be localized from naive to time zone-aware and converted from one time zone to another:

```
In [85]: stamp = pd.Timestamp('2011-03-12 04:00')
         stamp_utc = stamp.tz_localize('utc')
         stamp_utc.tz_convert('US/Eastern')
```

```
Out[85]: Timestamp('2011-03-11 23:00:00-0500', tz='US/Eastern')
```

You can also pass a time zone when creating the Timestamp.

```
In [86]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')
         stamp_moscow
```

```
Out[86]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

Time zone-aware Timestamp objects internally store a UTC timestamp value as nanoseconds since the UNIX epoch (January 1, 1970); this UTC value is invariant between time zone conversions:

```
In [87]: stamp_utc.value
```

```
Out[87]: 1299902400000000000
```

```
In [88]: stamp_utc.tz_convert('US/Eastern').value
```

```
Out[88]: 1299902400000000000
```

When performing time arithmetic using Pandas' DateOffset objects, daylight savings time transitions are respected where possible

```
In [89]: # 30 minutes before DST transition
         from pandas.tseries.offsets import Hour
         stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')
         stamp

Out[89]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')

In [90]: stamp + Hour()

Out[90]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')

In [91]: # 90 minutes before DST transition
         stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')
         stamp

Out[91]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')

In [92]: stamp + 2 * Hour()

Out[92]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

1.4.3 Operations between different time zones

If two time series with different time zones are combined, the result will be UTC. Since the timestamps are stored under the hood in UTC, this is a straightforward operation and requires no conversion to happen.

```
In [93]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')
         ts = Series(np.random.randn(len(rng)), index=rng)
         ts

Out[93]: 2012-03-07 09:30:00   -1.082743
         2012-03-08 09:30:00   -0.311357
         2012-03-09 09:30:00   -0.055951
         2012-03-12 09:30:00   -1.182513
         2012-03-13 09:30:00   -3.056361
         2012-03-14 09:30:00   -0.706622
         2012-03-15 09:30:00    0.196093
         2012-03-16 09:30:00    0.040397
         2012-03-19 09:30:00   -0.624963
         2012-03-20 09:30:00   -2.056198
         Freq: B, dtype: float64

In [94]: ts1 = ts[:7].tz_localize('Europe/London')
         ts2 = ts1[2:].tz_convert('Europe/Moscow')
         result = ts1 + ts2
         result.index

Out[94]: DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
                        '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
                        '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
                        '2012-03-15 09:30:00+00:00'],
                       dtype='datetime64[ns]', freq='B', tz='UTC')
```