

Lecture 07. Time Series

Instructor: Luping Yu

Apr 11, 2022

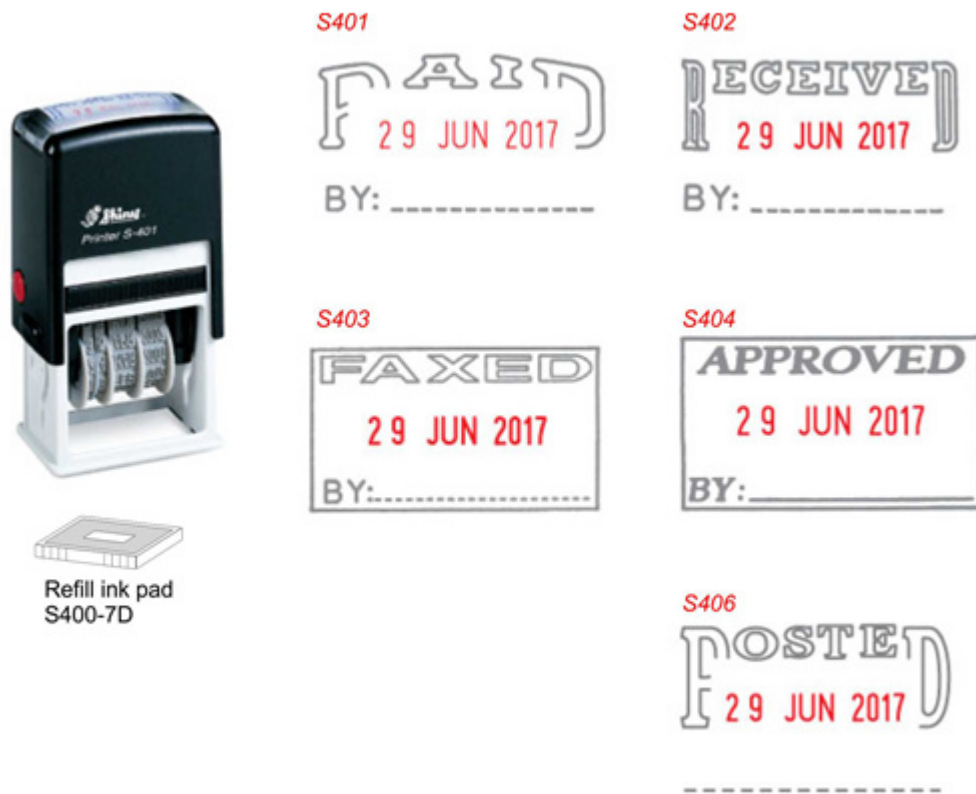
Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, and physics. Anything that is observed or measured at many points in time forms a time series.

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.

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)

The simplest and most widely used kind of time series are those indexed by **timestamp**.



Date and Time Data Types and Tools

The Python standard library includes data types for date and time data, as well as calendar-related functionality. The `datetime`, `time`, and `calendar` modules are the main places to start. The `datetime.datetime` type, or simply `datetime`, is widely used:

```
In [1]: from datetime import datetime

now = datetime.now()

now
```

```
Out[1]: datetime.datetime(2023, 4, 9, 22, 2, 11, 808890)
```

```
In [2]: now.year, now.month, now.day
```

```
Out[2]: (2023, 4, 9)
```

`datetime` stores both the date and time down to the microsecond. `timedelta` represents the temporal difference between two `datetime` objects:

```
In [3]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

delta
```

```
Out[3]: datetime.timedelta(days=926, seconds=56700)
```

```
In [4]: delta.days
```

```
Out[4]: 926
```

You can add (or subtract) a `timedelta` or multiple thereof to a `datetime` object to yield a new shifted object:

```
In [5]: from datetime import timedelta

start = datetime(2011, 1, 7)

start - timedelta(2)
```

```
Out[5]: datetime.datetime(2011, 1, 5, 0, 0)
```

```
In [6]: start + 2 * timedelta(weeks=2)
```

```
Out[6]: datetime.datetime(2011, 2, 4, 0, 0)
```

The following table summarizes the data types in the `datetime` module. While this chapter is mainly concerned with the data types in pandas and higher-level time series manipulation, you may encounter the datetime-based types in many other places in Python in the wild.

- Types in datetime module:

Type	Description
date	Store calendar date (year, month, day) using the Gregorian calendar
time	Store time of day as hours, minutes, seconds, and microseconds
datetime	Stores both date and time
timedelta	Represents the difference between twodatetimevalues (as days, seconds, and microseconds)
tzinfo	Base type for storing time zone information

Converting Between String and Datetime

You can format `datetime` objects and pandas `Timestamp` objects as strings using `str()` or the `strftime()` method, passing a format specification:

```
In [7]: stamp = datetime(2011, 1, 3)

str(stamp)
```

```
Out[7]: '2011-01-03 00:00:00'
```

You can use these same format codes to convert strings to dates using `datetime.strptime()`:

```
In [8]: value = '99:01:03'

datetime.strptime(value, '%y:%m:%d')
```

```
Out[8]: datetime.datetime(1999, 1, 3, 0, 0)
```

See the following table for a complete list of the format codes.

- Datetime format specification (ISO C89 compatible)

Type	Description
%Y	Four-digit year
%y	Two-digit year
%m	Two-digit month [01, 12]
%d	Two-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	Two-digit minute [00, 59]
%S	Second [00, 61] (seconds 60, 61 account for leap seconds)
%w	Weekday as integer [0 (Sunday), 6]
%U	Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0"
%W	Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are "week 0"
%z	UTC time zone offset as+HHMMor-HHMM; empty if time zone naive
%F	Shortcut for%Y-%m-%d (e.g.,2012-4-18)
%D	Shortcut for%m/%d/%y (e.g.,04/18/12)

`datetime.strptime` is a good way to parse a date with a **known** format.

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

```
Out[9]: datetime.datetime(2011, 1, 3, 0, 0)
```

`dateutil.parser` is capable of parsing most human-intelligible date representations:

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

```
Out[10]: datetime.datetime(1997, 1, 31, 22, 45)
```

`dateutil.parser` is a useful but **imperfect** tool. Notably, it will recognize some strings as dates that you might prefer that it didn't.

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

```
In [11]: parse('6/12/2011', dayfirst=True)
```

```
Out[11]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame.

The `to_datetime()` method parses many different kinds of date representations:

```
In [12]: import pandas as pd

datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

pd.to_datetime(datestrs)
```

```
Out[12]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'], dtype='datetime64[ns]', freq=None)
```

Time Series Basics

A basic kind of time series object in pandas is a **Series indexed by timestamps**:

```
In [13]: import numpy as np
import pandas as pd
```

```
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 = pd.Series(np.random.randn(6), index=dates)

ts
```

```
Out[13]: 2011-01-02    1.976024
         2011-01-05   -0.030445
         2011-01-07    1.271553
         2011-01-08   -0.302420
         2011-01-10   -1.120660
         2011-01-12   -0.030866
         dtype: float64
```

These **datetime** objects have been put in a **DatetimeIndex** :

```
In [14]: ts.index
```

```
Out[14]: DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
                        '2011-01-10', '2011-01-12'],
                        dtype='datetime64[ns]', freq=None)
```

Scalar values from a **DatetimeIndex** are pandas **Timestamp** objects. A Timestamp can be substituted anywhere you would use a datetime object.

```
In [15]: stamp = ts.index[0]

stamp
```

```
Out[15]: Timestamp('2011-01-02 00:00:00')
```

Indexing, Selection, Subsetting

Time series behaves like any other `pandas.Series` when you are indexing and selecting data based on label:

```
In [16]: ts
```

```
Out[16]: 2011-01-02    1.976024
         2011-01-05   -0.030445
         2011-01-07    1.271553
         2011-01-08   -0.302420
         2011-01-10   -1.120660
         2011-01-12   -0.030866
         dtype: float64
```

```
In [17]: stamp = ts.index[2]
```

```
stamp
```

```
Out[17]: Timestamp('2011-01-07 00:00:00')
```

```
In [18]: ts[stamp]
```

```
Out[18]: 1.271552641486588
```

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

```
In [19]: ts
```

```
Out[19]: 2011-01-02    1.976024
         2011-01-05   -0.030445
         2011-01-07    1.271553
         2011-01-08   -0.302420
         2011-01-10   -1.120660
         2011-01-12   -0.030866
         dtype: float64
```

```
In [20]: ts['Jan/07/2011']
```

```
Out[20]: 1.271552641486588
```



```
In [21]: ts['20110107']
```

```
Out[21]: 1.271552641486588
```

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

```
In [22]: longer_ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
```

```
longer_ts
```

```
Out[22]: 2000-01-01    -0.953493
          2000-01-02     0.121024
          2000-01-03     1.625070
          2000-01-04     0.942599
          2000-01-05     0.950422
          ...
          2002-09-22    -1.724148
          2002-09-23     0.350723
          2002-09-24    -0.241075
          2002-09-25     1.271666
          2002-09-26     0.142564
          Freq: D, Length: 1000, dtype: float64
```

```
In [23]: longer_ts['2001']
```

```
Out[23]: 2001-01-01     0.887724
          2001-01-02    -0.896955
          2001-01-03     0.562308
          2001-01-04     0.947738
          2001-01-05     0.866262
          ...
          2001-12-27     0.661008
          2001-12-28     1.457987
          2001-12-29    -0.581081
          2001-12-30     0.307636
          2001-12-31    -0.132754
          Freq: D, Length: 365, dtype: float64
```

Here, the string '2001' is interpreted as a year and selects that time period. This also works if you specify the month:

```
In [24]: longer_ts['2001-05']
```

```
Out[24]: 2001-05-01    -1.075826
          2001-05-02    -0.368523
          2001-05-03    -0.724405
          2001-05-04    -0.678176
          2001-05-05     1.902710
          2001-05-06    -0.088312
          2001-05-07    -0.205817
          2001-05-08     1.168976
          2001-05-09     0.276810
          2001-05-10     0.661231
          2001-05-11    -1.114754
          2001-05-12     0.210153
          2001-05-13     0.241280
          2001-05-14     0.925500
          2001-05-15    -0.463700
          2001-05-16    -1.063321
          2001-05-17     0.405435
          2001-05-18    -0.868674
          2001-05-19    -0.814850
          2001-05-20     0.035860
          2001-05-21     0.495335
          2001-05-22     0.095867
          2001-05-23     1.667603
          2001-05-24     0.007068
          2001-05-25    -0.626161
          2001-05-26    -0.304650
          2001-05-27    -1.162671
          2001-05-28     0.466192
          2001-05-29     0.584128
          2001-05-30     1.317576
          2001-05-31     0.317257
          Freq: D, dtype: float64
```

Slicing with datetime objects works as well:

```
In [25]: ts
```

```
Out[25]: 2011-01-02    1.976024
         2011-01-05   -0.030445
         2011-01-07    1.271553
         2011-01-08   -0.302420
         2011-01-10   -1.120660
         2011-01-12   -0.030866
         dtype: float64
```

```
In [26]: ts['1/7/2011':]
```

```
Out[26]: 2011-01-07    1.271553
         2011-01-08   -0.302420
         2011-01-10   -1.120660
         2011-01-12   -0.030866
         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 [27]: ts['Jan/6/2011':'1/11/2011']
```

```
Out[27]: 2011-01-07    1.271553
         2011-01-08   -0.302420
         2011-01-10   -1.120660
         dtype: float64
```

As before, you can pass **either** a string date, datetime, or timestamp.

All of this holds true for `DataFrame` as well, indexing on its rows:

```
In [28]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')

         long_df = pd.DataFrame(np.random.randn(100, 4),
                                index=dates,
                                columns=['Beijing', 'Shanghai', 'Guangdong', 'Fujian'])

         long_df
```

Out[28]:

	Beijing	Shanghai	Guangdong	Fujian
2000-01-05	1.636802	-0.133292	-1.231817	1.797256
2000-01-12	0.609883	0.292168	1.576165	-0.511653
2000-01-19	-0.898964	-0.189264	-1.185821	-0.916561
2000-01-26	-1.940313	0.161989	-0.080756	-0.659727
2000-02-02	-0.474919	-0.541740	0.144476	-1.843400
...
2001-10-31	0.233839	0.177758	-0.270259	-0.288995
2001-11-07	-0.508430	0.261283	0.819676	1.150036
2001-11-14	-1.025691	-0.068338	0.407646	-0.162213
2001-11-21	-0.173569	-0.004976	-0.054656	-0.519738
2001-11-28	-0.719963	-0.141201	1.248739	-0.314746

100 rows x 4 columns

```
In [29]: long_df.loc['2001----May']
```

Out[29]:

	Beijing	Shanghai	Guangdong	Fujian
2001-05-02	0.131639	-0.943033	-1.118936	1.249002
2001-05-09	1.025719	0.822297	0.371603	0.185495
2001-05-16	1.934397	-0.009675	-0.765639	0.935663
2001-05-23	-0.370031	-0.791216	-0.723535	-1.704712
2001-05-30	-1.889028	0.885304	0.055354	0.527706

Date Ranges, Frequencies, and Shifting

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 every 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**.

While we used it previously without explanation, `pandas.date_range` is responsible for generating a `DatetimeIndex` with an indicated length according to a particular frequency:

```
In [30]: index = pd.date_range('2012-04-01', '2012-06-01')
```

```
index
```

```
Out[30]: 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')
```

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 [31]: pd.date_range(start='2012-04-01', periods=20)
```

```
Out[31]: 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')
```

```
In [32]: pd.date_range(end='2012-06-01', periods=20)
```

```
Out[32]: 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')
```

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; see more complete listing of frequencies in the following table) and only dates falling on or inside the date interval will be included:

```
In [33]: pd.date_range('2000-01-01', '2000-12-01', freq='BM')
```

```
Out[33]: 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')
```

- Base time series frequencies (not comprehensive)

Alias	Offset type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely

Alias	Offset type	Description
S	Second	Secondly
L or ms	Milli	Millisecond (1/1,000 of 1 second)
U	Micro	Microsecond (1/1,000,000 of 1 second)
M	MonthEnd	Last calendar day of month
BM	BusinessMonthEnd	Last business day (weekday) of month
MS	MonthBegin	First calendar day of month
BMS	BusinessMonthBegin	First weekday of month
W-MON, W-TUE, ...	Week	Weekly on given day of week (MON, TUE, WED, THU, FRI, SAT, or SUN)
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month (e.g., WOM-3FRI for the third Friday of each month)
Q-JAN, Q-FEB, ...	QuarterEnd	Quarterly dates anchored on last calendar day of each month, for year ending in indicated month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Quarterly dates anchored on last weekday day of each month, for year ending in indicated month
QS-JAN, QS-FEB, ...	QuarterBegin	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Quarterly dates anchored on first weekday day of each month, for year ending in indicated month
A-JAN, A-FEB, ...	YearEnd	Annual dates anchored on last calendar day of given month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
BA-JAN, BA-FEB, ...	BusinessYearEnd	Annual dates anchored on last weekday of given month
AS-JAN, AS-FEB, ...	YearBegin	Annual dates anchored on first day of given month
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Annual dates anchored on first weekday of given month

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. Putting an integer before the base frequency creates a multiple:

```
In [34]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4H')
```

```
Out[34]: 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')
```

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 whether the month ends on a weekend or not. We refer to these as **anchored offsets**.

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 [35]: third_friday = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')
```

```
third_friday
```

```
Out[35]: DatetimeIndex(['2012-01-20', '2012-02-17', '2012-03-16', '2012-04-20',
                        '2012-05-18', '2012-06-15', '2012-07-20', '2012-08-17'],
                        dtype='datetime64[ns]', freq='WOM-3FRI')
```

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

ts
```

```
Out[36]: 2000-01-31    -0.606474
         2000-02-29    -0.681297
         2000-03-31     1.738421
         2000-04-30     1.040436
         Freq: M, dtype: float64
```

```
In [37]: ts.shift(2)
```

```
Out[37]: 2000-01-31         NaN
         2000-02-29         NaN
         2000-03-31    -0.606474
         2000-04-30    -0.681297
         Freq: M, dtype: float64
```

```
In [38]: ts.shift(-2)
```

```
Out[38]: 2000-01-31     1.738421
         2000-02-29     1.040436
         2000-03-31         NaN
         2000-04-30         NaN
         Freq: M, dtype: float64
```

When we shift like this, missing data is introduced either at the start or the end of the time series.

A common use of shift is computing **percent changes in a time series**:

```
In [39]: ts
```

```
Out[39]: 2000-01-31    -0.606474
         2000-02-29    -0.681297
         2000-03-31     1.738421
         2000-04-30     1.040436
         Freq: M, dtype: float64
```

```
In [40]: ts / ts.shift(1) - 1
```

```
Out[40]: 2000-01-31      NaN
         2000-02-29      0.123374
         2000-03-31     -3.551636
         2000-04-30     -0.401505
         Freq: M, dtype: float64
```

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 [41]: ts
```

```
Out[41]: 2000-01-31     -0.606474
         2000-02-29     -0.681297
         2000-03-31      1.738421
         2000-04-30      1.040436
         Freq: M, dtype: float64
```

```
In [42]: ts.shift(2, freq='M')
```

```
Out[42]: 2000-03-31     -0.606474
         2000-04-30     -0.681297
         2000-05-31      1.738421
         2000-06-30      1.040436
         Freq: M, dtype: float64
```

```
In [43]: ts.shift(3, freq='D')
```

```
Out[43]: 2000-02-03     -0.606474
         2000-03-03     -0.681297
         2000-04-03      1.738421
         2000-05-03      1.040436
         dtype: float64
```

Periods and Period Arithmetic

Periods represent timespans, like days, months, quarters, or years. The `Period` class represents this data type, requiring a string or integer and a frequency.

```
In [44]: p = pd.Period(2007, freq='A-DEC')  
  
p
```

```
Out[44]: Period('2007', 'A-DEC')
```

In this case, the Period object represents the full timespan from January 1, 2007, to December 31, 2007, inclusive.

Period Frequency Conversion

Periods and PeriodIndex objects can be converted to another frequency with their `asfreq()` method.

As an example, suppose we had an annual period and wanted to convert it into a monthly period either at the start or end of the year. This is fairly straightforward:

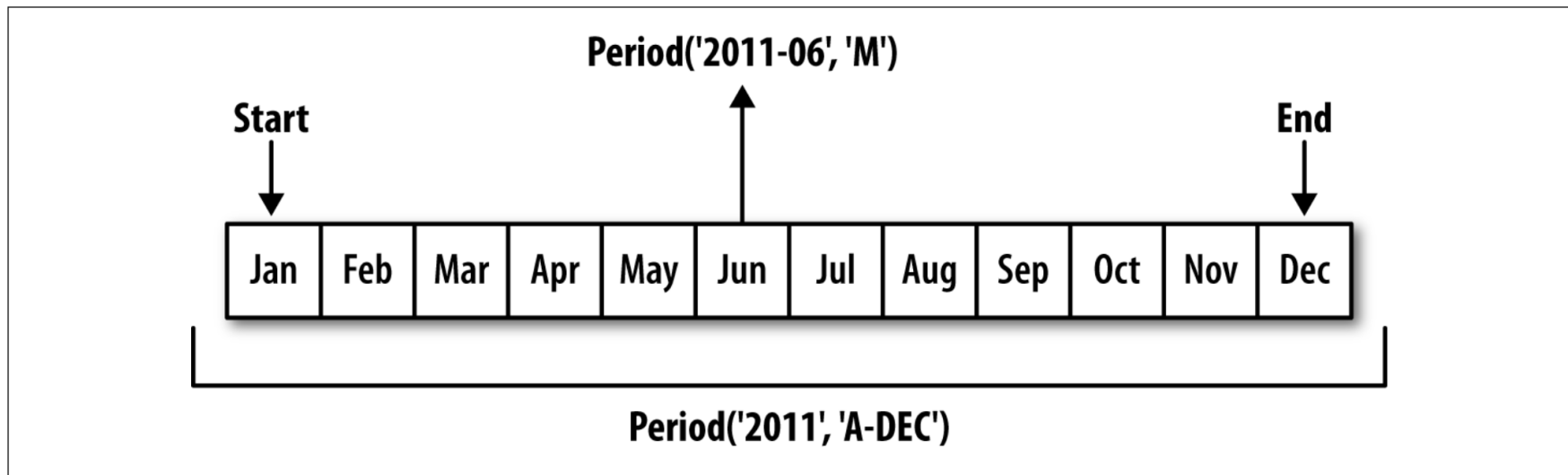
```
In [45]: p.asfreq('M', how='start')
```

```
Out[45]: Period('2007-01', 'M')
```

```
In [46]: p.asfreq('M', how='end')
```

```
Out[46]: Period('2007-12', 'M')
```

- Period frequency conversion illustration



Quarterly Period Frequencies

Quarterly data is standard in accounting, finance, and other fields. Much quarterly data is reported relative to a **fiscal year end**, typically the last calendar or business day of one of the 12 months of the year.

pandas supports all 12 possible quarterly frequencies as Q-JAN through Q-DEC:

```
In [47]: p = pd.Period('2012Q4', freq='Q-JAN')
```

```
p
```

```
Out[47]: Period('2012Q4', 'Q-JAN')
```

In the case of fiscal year ending in January, 2012Q4 runs from November through January, which you can check by converting to daily frequency:

```
In [48]: p.asfreq('D', 'start')
```

```
Out[48]: Period('2011-11-01', 'D')
```

```
In [49]: p.asfreq('D', 'end')
```

```
Out[49]: Period('2012-01-31', 'D')
```

- Different quarterly frequency conventions

Year 2012													
M	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC	
Q-DEC	2012Q1			2012Q2			2012Q3			2012Q4			
Q-SEP	2012Q2			2012Q3			2012Q4			2013Q1			
Q-FEB	2012Q4			2013Q1			2013Q2			2013Q3			Q4

Resampling and Frequency Conversion

Resampling refers to the process of converting a time series from one frequency to another.

Aggregating higher frequency data to lower frequency is called **downsampling**, while converting lower frequency to higher frequency is called **upsampling**.

pandas objects are equipped with a `resample()` method, which is the workhorse function for all frequency conversion.

`resample()` has a similar API to `groupby()` ; you call `resample` to group the data, then call an aggregation function:

```
In [50]: rng = pd.date_range('2000-01-01', periods=100, freq='D')
```

```
ts = pd.Series(np.random.randn(len(rng)), index=rng)
```

```
ts
```

```
Out [50]: 2000-01-01    0.954870
          2000-01-02    0.109680
          2000-01-03   -0.050917
          2000-01-04    0.247322
          2000-01-05    2.024514
          ...
          2000-04-05   -1.152745
          2000-04-06    1.721990
          2000-04-07   -0.199907
          2000-04-08    0.362957
          2000-04-09    0.957613
          Freq: D, Length: 100, dtype: float64
```

```
In [51]: ts.resample('M').mean()
```

```
Out [51]: 2000-01-31    0.396260
          2000-02-29   -0.138347
          2000-03-31   -0.199954
          2000-04-30    0.034279
          Freq: M, dtype: float64
```

```
In [52]: ts.resample('M', kind='period').mean()
```

```
Out [52]: 2000-01    0.396260
          2000-02   -0.138347
          2000-03   -0.199954
          2000-04    0.034279
          Freq: M, dtype: float64
```

`resample()` is a flexible and high-performance method that can be used to process very large time series. The following table summarizes some of its options.

- Resample method arguments

Argument	Description
freq	String or DateOffset indicating desired resampled frequency (e.g., 'M', '5min', or Second(15))
axis	Axis to resample on; default axis=0
fill_method	How to interpolate when upsampling, as in 'ffill' or 'bfill'; by default does no interpolation
closed	In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'
label	In downsampling, how to label the aggregated result, with the 'right' or 'left' bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled 9:30 or 9:35)
loffset	Time adjustment to the bin labels, such as '-1s' / Second(-1) to shift the aggregate labels one second earlier
limit	When forward or backward filling, the maximum number of periods to fill
kind	Aggregate to periods ('period') or timestamps ('timestamp'); defaults to the type of index the time series has
convention	When resampling periods, the convention ('start' or 'end') for convert

Downsampling

Aggregating data to a regular, lower frequency is a pretty normal time series task.

The desired frequency defines bin edges that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, you need to chop up the data into one-month intervals.

Each interval is said to be **half-open**; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame.

To illustrate, let's look at some one-minute data:

```
In [53]: rng = pd.date_range('2000-01-01', periods=12, freq='T')
ts = pd.Series(np.arange(12), index=rng)
ts
```

```
Out [53]: 2000-01-01 00:00:00    0
          2000-01-01 00:01:00    1
          2000-01-01 00:02:00    2
          2000-01-01 00:03:00    3
          2000-01-01 00:04:00    4
          2000-01-01 00:05:00    5
          2000-01-01 00:06:00    6
          2000-01-01 00:07:00    7
          2000-01-01 00:08:00    8
          2000-01-01 00:09:00    9
          2000-01-01 00:10:00   10
          2000-01-01 00:11:00   11
          Freq: T, dtype: int64
```

Suppose you wanted to aggregate this data into five-minute chunks or bars by taking the sum of each group:

```
In [54]: ts.resample('5min').sum()
```

```
Out [54]: 2000-01-01 00:00:00    10
          2000-01-01 00:05:00    35
          2000-01-01 00:10:00    21
          Freq: 5T, dtype: int64
```

Upsampling and Interpolation

When converting from a low frequency to a higher frequency, no aggregation is needed.

Let's consider a DataFrame with some weekly data:

```
In [55]: frame = pd.DataFrame(np.random.randn(2, 4),
                              index=pd.date_range('1/1/2000', periods=2, freq='W-WED'),
                              columns=['Colorado', 'Texas', 'New York', 'Ohio'])

frame
```



```
Out [55]:
```

	Colorado	Texas	New York	Ohio
2000-01-05	-1.681419	-0.233672	0.307125	0.341754
2000-01-12	0.391169	-0.006416	1.638438	-0.028422

When you are using an aggregation function with this data, there is only one value per group, and missing values result in the gaps.

We use the `asfreq` method to convert to the higher frequency without any aggregation:

```
In [56]: df_daily = frame.resample('D').asfreq()

df_daily
```

```
Out [56]:
```

	Colorado	Texas	New York	Ohio
2000-01-05	-1.681419	-0.233672	0.307125	0.341754
2000-01-06	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	0.391169	-0.006416	1.638438	-0.028422

Suppose you wanted to fill forward each weekly value on the non-Wednesdays:

```
In [57]: frame.resample('D').ffill()
```

Out [57]:

	Colorado	Texas	New York	Ohio
2000-01-05	-1.681419	-0.233672	0.307125	0.341754
2000-01-06	-1.681419	-0.233672	0.307125	0.341754
2000-01-07	-1.681419	-0.233672	0.307125	0.341754
2000-01-08	-1.681419	-0.233672	0.307125	0.341754
2000-01-09	-1.681419	-0.233672	0.307125	0.341754
2000-01-10	-1.681419	-0.233672	0.307125	0.341754
2000-01-11	-1.681419	-0.233672	0.307125	0.341754
2000-01-12	0.391169	-0.006416	1.638438	-0.028422

You can similarly choose to only fill a certain number of periods forward to limit how far to continue using an observed value:

```
In [58]: frame.resample('D').ffill(limit=2)
```

Out [58]:

	Colorado	Texas	New York	Ohio
2000-01-05	-1.681419	-0.233672	0.307125	0.341754
2000-01-06	-1.681419	-0.233672	0.307125	0.341754
2000-01-07	-1.681419	-0.233672	0.307125	0.341754
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	0.391169	-0.006416	1.638438	-0.028422

Moving Window Functions

Before digging in, we can load up some time series data and resample it to business day frequency:

```
In [59]: import pandas as pd

close_px_all = pd.read_csv('examples/stock_px_2.csv',
                           parse_dates=True, index_col=0)

close_px_all
```

```
Out[59]:
```

	AAPL	MSFT	XOM	SPX
2003-01-02	7.40	21.11	29.22	909.03
2003-01-03	7.45	21.14	29.24	908.59
2003-01-06	7.45	21.52	29.96	929.01
2003-01-07	7.43	21.93	28.95	922.93
2003-01-08	7.28	21.31	28.83	909.93
...
2011-10-10	388.81	26.94	76.28	1194.89
2011-10-11	400.29	27.00	76.27	1195.54
2011-10-12	402.19	26.96	77.16	1207.25
2011-10-13	408.43	27.18	76.37	1203.66
2011-10-14	422.00	27.27	78.11	1224.58

2214 rows × 4 columns

```
In [60]: close_px = close_px_all.resample('B').ffill()

close_px
```

Out[60]:

	AAPL	MSFT	XOM	SPX
2003-01-02	7.40	21.11	29.22	909.03
2003-01-03	7.45	21.14	29.24	908.59
2003-01-06	7.45	21.52	29.96	929.01
2003-01-07	7.43	21.93	28.95	922.93
2003-01-08	7.28	21.31	28.83	909.93
...
2011-10-10	388.81	26.94	76.28	1194.89
2011-10-11	400.29	27.00	76.27	1195.54
2011-10-12	402.19	26.96	77.16	1207.25
2011-10-13	408.43	27.18	76.37	1203.66
2011-10-14	422.00	27.27	78.11	1224.58

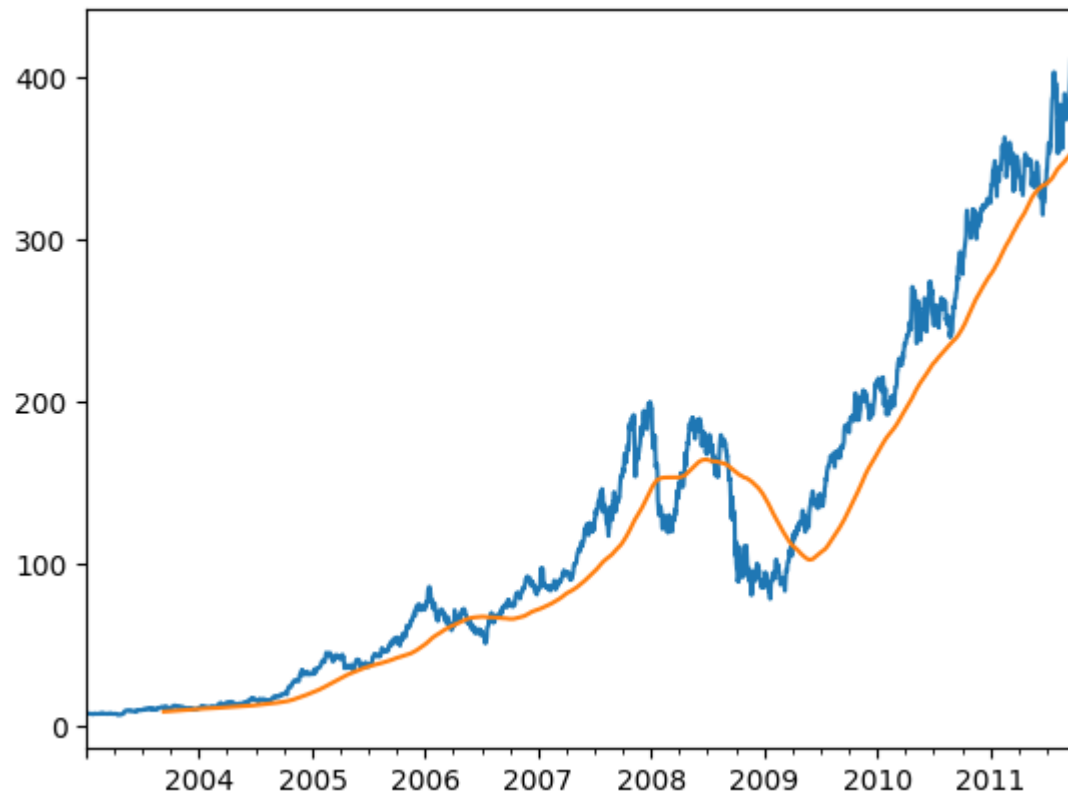
2292 rows × 4 columns

We now introduce the `rolling()` operator, which behaves similarly to `resample()` and `groupby()`. It can be called on a Series or DataFrame along with a window:

```
In [61]: %matplotlib inline
          %config InlineBackend.figure_format = 'svg'

          close_px['AAPL'].plot()
          close_px['AAPL'].rolling(180).mean().plot()
```

Out[61]: <Axes: >



Some statistical operators, like correlation and covariance, need to operate on two time series.

As an example, financial analysts are often interested in a stock's correlation to a benchmark index like the S&P 500. To have a look at this, we first compute the percent change for all of our time series of interest:

```
In [62]: returns = close_px / close_px.shift(1) - 1  
  
returns
```

Out[62]:

	AAPL	MSFT	XOM	SPX
2003-01-02	NaN	NaN	NaN	NaN
2003-01-03	0.006757	0.001421	0.000684	-0.000484
2003-01-06	0.000000	0.017975	0.024624	0.022474
2003-01-07	-0.002685	0.019052	-0.033712	-0.006545
2003-01-08	-0.020188	-0.028272	-0.004145	-0.014086
...
2011-10-10	0.051406	0.026286	0.036977	0.034125
2011-10-11	0.029526	0.002227	-0.000131	0.000544
2011-10-12	0.004747	-0.001481	0.011669	0.009795
2011-10-13	0.015515	0.008160	-0.010238	-0.002974
2011-10-14	0.033225	0.003311	0.022784	0.017380

2292 rows × 4 columns

The `corr` aggregation function after we call rolling can then compute the rolling correlation with S&P Index:

```
In [63]: corr = returns['AAPL'].rolling(125).corr(returns['SPX'])  
corr.plot(title="Six-month AAPL return correlation to S&P 500")
```

```
Out[63]: <Axes: title={'center': 'Six-month AAPL return correlation to S&P 500'}>
```



```
In [64]: corr = returns.rolling(125).corr(returns['SPX'])  
corr.plot(title="Six-month return correlations to S&P 500")
```

```
Out[64]: <Axes: title={'center': 'Six-month return correlations to S&P 500'}>
```

Six-month return correlations to S&P 500

