시계열 다루기

Pandas는 금융 모델링을 위해 개발돼 예상대로 날짜, 시간, 시간 인덱스를 가진 데이터를 다루는 매우 다양한 도구를 갖고 있다. 날짜와 시간 데이터에는 다음과 같이 몇가지 종류가 있다.

- 타임스템프 : 특정 시점을 말한다.(2015년 6월 4일 오전 7:00).
- 시간 간격과 기간 : 특정 시작점과 종료 점 사이의 시간의 길이를 말한다. 예를 들어 2015년이 여기에 해당한다. 기간은 일반적으로 각 간격이 일정하고 서로 겹치지 않는 특별한 경우의 시간 간격을 말한다. (예: 하루를 구성하는 24시간이라는 기간)
- 시간 델타(time-delta)나 지속시간(duration)은 정확한 시간 길이를 말한다.(예, 22.56초).

Pandas의 각 날짜/시간 유형으로 작업하는 법을 소개한다. Pandas가 제공하는 도구에 대해 더 구체적으로 알아보기 전에 우선 파이썬에서 날짜와 시간을 처리하는 도구에 대해서 알아본다.

파이썬에서의 날짜와 시간

파이썬 세계에는 여러가지 날짜, 시간, 델타, 시간 간격을 표현하는 다양한 방식이 있다.
Pandas가 제공하는 시계열 도구는 데이터 과학 애플리메이션에서 가장 유용하지만, 파이썬에서 사용하는 다른 패키지와의 관계를 알아보는 것이 도움이 된다.

기본Python 날짜와 시간 : datetime 과 dateutil

날자와 시간으로 작업하는 파이썬 기본 개체는 내장 모듈인 datatime에 존재한다. 제3자 모듈인 datautil과 함께 datatime모듈을 사용해 날짜와 시간에 여러 가지 유용한 기능을 신속하게 수행할 수 있다. 예를 들어 datetime타입을 사용해 날짜를 직접 구성할 수 있다.

```
In [1]:from datetime import datetime
       datetime(year=2015, month=7, day=4)
Out[1]:datetime.datetime(2015, 7, 4, 0, 0)
```

또는 dateUtil 모듈을 이용해 다양한 문자열 형태로 부터 날짜를 해석할 수 있다.

```
In [2]:from dateutil import parser
       date = parser.parse("4th of July, 2015")
       date
Out[2]:datetime.datetime(2015, 7, 4, 0, 0)
```

Datetime 객체를 갖고 나면 요일을 출력하는 등의 작업을 할 수 있다.:

```
In [3]:date.strftime('%A')
Out[3]:'Saturday'
```

마지막 줄에서 날짜를 출력하기 위한 표준 문자열 포맷 코드 중 하나("%A")를 사용했는데, 이에 대해서는 파이썬 datetime 문서의 strtime절에서 확인할 수 있다. 다른 유용한 날짜 유틸리티에 대한 내용은 dateutil온라인 문서에서 찾을 수 있다. 관련패키지로 pytz가 있는데, 이 패키지는 시계열 데이터(시간대)의 가장 골치 아픈 부분을 작업하기 위한 도구가 들어 있다.

Datetime과 dateutil의 능력은 유연성과 쉬운 구문에 있다. 관심이 있을 만한 거의 모든 연산은 이 객체와 그 내장된 메서드를 사용해 쉽게 수행할 수 있다. 문제는 큰 규모의 날짜와 시간 배열로 작업할 때 발생한다. NumPy 스타일 타입이 지정된 숫자 배열이 파이썬 숫자 변수 리스트보다 나은 것처럼 인코딩 된 날짜의 타입 지정 배열이 파이썬 datetime 객체의 리스트보다 낮다.

타입이 지정된 시간 배열 : NumPy의 datetime64

파이썬의 datetime포맷의 약점에서 영감을 받아 NumPy팀은 NumPy에 몇가지 기본 시계열 데이터 타입을 추가했다. Datetime64 dtype은 날짜를 64비트 정수로 인코딩해 날짜 배열을 매우 간결하게 표현하게 해준다. Datetime는 매우 구체적인 입력 형식이 필요하다.

```
In [4]:import numpy as np
        date = np.array('2015-07-04', dtype=np.datetime64)
        date
Out[4]:array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

하지만 이 날짜 포맷이 정해지고 나면 거기에 벡터화된 연산을 빠르게 수행할 수 있다.:

```
In [5]:date + np.arange(12)
Out[5]:array(['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[D]')
```

NumPy datetime64 배열은 하나의 타입을 가지고 있기 때문에 이 유형의 연산이 파이썬의 datetime 객체로 직접 작업하는 것보다 훨씬 더 빨리 수행될 수 있다. 특히 배열 크기가 커질 수록 더 빠르다.
Datetime64와 timedelta64객체의 세부사항 중 하나는 기본 시간 단위 기반으로 만들어졌다는 것이다. Datetime64객체는 64비트 정밀도에 제한되기 때문에 인코딩할 수 있는 시간의 범위가 이 기본 단위의 배다, 다시말해, daetime64는 시간 분해능(time resolution)과 시간사이의 절충점을 도입한다,
가령 시간 분해를 1나노초 단위로 하고 싶다면 최대 나노초 또는 600년 니하로만 인코딩할 수 있다. NumPy는 입력값으로부터 원하는 단위를 추론한다,

하루단위의 datetime

```
In [6]:np.datetime64('2015-07-04')
Out[6]:numpy.datetime64('2015-07-04')
```

분단위의 datetime

```
In [7]:np.datetime64('2015-07-04 12:00')
Out[7]:numpy.datetime64('2015-07-04T12:00')
```

시간대는 코드를 실행하는 컴퓨터의 지역 시간으로 자동으로 설정된다는 사실에 주목하자. 다양한 포맷코드 중 하나를 사용해 원하는 기본 단위를 정할 수 있다.

시간의 단위를 나노초로 정한 것:

```
In [8]:np.datetime64('2015-07-04 12:59:59.50', 'ns')
Out[8]:numpy.datetime64('2015-07-04T12:59:59.500000000')
```

NumPy datetime64문서에서 가져온 사용 가능한 포맷 코드와 NumPy datetime64가 인코딩할 수 있는 상대적이자 절대적인 시간 범위를 정리한 것이다.:

Code	Meaning	Time span (relative)	Time span (absolute)
Y	Year	± 9.2e18 years	[9.2e18 BC, 9.2e18 AD]
M	Month	± 7.6e17 years	[7.6e17 BC, 7.6e17 AD]
W	Week	± 1.7e17 years	[1.7e17 BC, 1.7e17 AD]
D	Day	± 2.5e16 years	[2.5e16 BC, 2.5e16 AD]
h	Hour	± 1.0e15 years	[1.0e15 BC, 1.0e15 AD]

m	Minute	± 1.7e13 years	[1.7e13 BC, 1.7e13 AD]
s	Second	± 2.9e12 years	[2.9e9 BC, 2.9e9 AD]
ms	Millisecond	± 2.9e9 years	[2.9e6 BC, 2.9e6 AD]
us	Microsecond	± 2.9e6 years	[290301 BC, 294241 AD]
ns	Nanosecond	± 292 years	[1678 AD, 2262 AD]
ps	Picosecond	± 106 days	[1969 AD, 1970 AD]
fs	Femtosecond	± 2.6 hours	[1969 AD, 1970 AD]
as	Attosecond	± 9.2 seconds	[1969 AD, 1970 AD]

현실 세계의 데이터 타입인 경우 유용한 기본값은 datetime64[ns] 이다. 이 값이 적절한 정밀도로 현대 날짜의 유용한 범위를 인코딩할 수 있기 때문이다
마지막으로 datetime64 데이트 타입은 파이썬의 내장 타입인 datetime의 결함을 어느 정도 해결해 주지만, datetime과 특히 dateutil이 제공하는 여러 가지 편리한 메서드와 함수가 없다,

pandas에서의 날짜와 시간: 두 세계의 최선

Pandas는 Timestamp 객체를 제공하기 위해 방금 논의했던 모든 도구를 기반으로 만들어졌다, 그 객체는 datetime과 dateutil의 사용 편리성과 numPy.datetime64의 효율적인 저장소와 벡터화된 인터페이스를 지정하는 데 사용할 수 있는 DatetimeIndex를 출력할 수 있다.

Pandas 도구를 사용해 앞에서 보여준 코드를 다시 구현할 수 있다. 다양한 포맷을 사용하는 문자열 날짜 데이터를 해석하고 포맷 코드를 이용해 요일을 출력할 수 있다.

```
In [9]:import pandas as pd
       date = pd.to_datetime("4th of July, 2015")
       date
Out[9]:Timestamp('2015-07-04 00:00:00')
In [10]:date.strftime('%A')
Out[10]:'Saturday'
```

게다가 같은 객체에 NumPy스타일의 벡터화된 연산을 직접 수행할 수도 있다.:

```
In [11]:date + pd.to_timedelta(np.arange(12), 'D')
Out[11]: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)
```

Pandas 시계열: 시간을 인덱싱하기

Pandas 시계열 도구는 실제로 타임스탬프로 데이터를 인덱싱할 때 아주 유용하다.

시간 인덱스르 가진 데이터의 Series객체를 구성할 수 있다.

```
In [12]:index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                                  '2015-07-04', '2015-08-04'])
         data = pd.Series([0, 1, 2, 3], index=index)
         data
```

```
Out[12]:2014-07-04    0
        2014-08-04    1
        2015-07-04    2
        2015-08-04    3
        dtype: int64
```

이 데이터를 Series에 저장했으니 이제 날짜로 변환될 수 있는 값을 전달해 모든 Series인덱싱 패턴에 사용할 수 있다.:

```
In [13]:data['2014-07-04':'2015-07-04']
Out[13]:2014-07-04    0
        2014-08-04    1
        2015-07-04    2
        dtype: int64
```

그 밖에도 해당 연도의 모든 데이터 슬라이스를 얻기 위해 연도를 전달하는 것과 같은 특별한 날짜 전용 인덱싱 연산이 있다.:

```
In [14]:data['2015']
Out[14]:2015-07-04    2
        2015-08-04    3
        dtype: int64
```

Pandas 시계열 데이터 구조

시계열 데이터 작업을 위한 기본적인 Pandas 데이터 구조를 소개한다

- 타임스템프(timestamp) : Pandas는 Timestamp타입을 제공한다, 앞에서 언급했듯이 이것은 근본적으로 파이썬의 기본 datetime의 대체 타입이지만 좀 더 효율적인 NumPy.datetime64 데이터 타입을 기반으로 한다. 관련 인덱스 구조는 datetimeIndex
- 기간(time period)의 경우 : Pandas는 Period타입을 제공한다. Numpy.datetime64를 기반으로 고정 주파수 간격을 인코딩한다. 관련 인덱스 구조는 PeriodIndex다
- 시간 델타 또는 지속 기간의 경우 : Pandas는 timestamp 타입을 제공한다. Timedelta는 파이썬 기본 datetime.timedelda타입의 좀 더 효율적인 대체 타입이며 numpy.timedelta64를 기반으로 한다. 관련 인덱스 구조는 TimedeltaIndex다,

이 날짜/시간 객체의 가장 기본은 Timestamp와 DatetimeIndex객체이다. 이 클래스 객체는 직접 호출될 수 있지만, 다양한 형식을 분석할 수 있는 pd.to_datetime()함수를 사용하는 것이 더 일반적이다. 단일 날짜를 pd.to_datetime()에 전달하면 Timestamp를 생성하고, 일련의 날짜를 전달하면 DatetimeIndex를 생성하는 것이 기본이다.,

```
In [15]:dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                                '2015-Jul-6', '07-07-2015', '20150708'])
        dates
Out[15]:DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                                                  '2015-07-08'],
                      dtype='datetime64[ns]', freq=None)
```

DatetimeIndex는 to_period()함수에 주기(frequency)코드를 추가해 PeriodIndex로 전달할 수 있다. 여기서는 일별 주기를 가르키는 'D'를 사용할 것 이다.:

```
In [16]:dates.to_period('D')
Out[16]:PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                                   '2015-07-08'],
```

```
                              dtype='int64', freq='D')
```

어떤 날짜에서 다른 날짜를 빼면 TimedeltaIndex가 생성된다r:

```
In [17]:dates - dates[0]
Out[17]:TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'], dtype='timedelta64[ns]', freq=None)
```

정규 시퀀스: pd.date_range()¶

Pandas는 정규 날짜 시퀀스를 더 편리하게 만들 수 있도록 몇 가지 함수를 제공한다. 타임 스탬프를 위한 pd.date_range(), 기간을 위한 pd.period._range(), 시간델타를 위한 pd.timedelta_range()가 여기에 해당한다.
파이썬의 range()함수와 NumPy의 np.arang()는 시작점, 종료점, 선택적 간격을 시퀀스로 전환한다. 마찬가지로 pd.date_range()는 시작일, 종료일, 선택적 주기 코드를 받아서 정규 날짜 시퀀스를생성한다.
기본적으로 주기는 하루로 설정돼 있다.

```
In [18]:pd.date_range('2015-07-03', '2015-07-10')
Out[18]: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')
```

이와 다르게 날짜 범위를 시작점과 종료점이 아니라 시작점과 기간의 수로 지정할 수도 있다.

```
In [19]:pd.date_range('2015-07-03', periods=8)
Out[19]: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')
```

freq인수를 바꿔서 간격을 조정할 수 있는데, 기본값은 0로 설정돼 있다.
시간 단위의 타임스탬프 범위를 만들어 보자.

```
In [20]:pd.date_range('2015-07-03', periods=8, freq='H')
Out[20]:DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
                       '015-07-03 02:00:00', '2015-07-03 03:00:00',
                       '2015-07-03 04:00:00', '2015-07-03 05:00:00',
                       '2015-07-03 06:00:00', '2015-07-03 07:00:00'],
                      dtype='datetime64[ns]', freq='H')
```

이와 매우 유사한 pd.period_range()와 pd.timedelta_range()함수는 기간이나 타임 텔타값의 정규 시퀀스를 생성하는데 유용하다.
월 단위 기간을 예로 든 것이다.

```
In [21]:pd.period_range('2015-07', periods=8, freq='M')
Out[21]:PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
                     '2016-01', '2016-02'],
                    dtype='int64', freq='M')
```

시간 단위로 증가하는 기간의 시퀀스를 생성한 것이다.

```
In [22]:pd.timedelta_range(0, periods=10, freq='H')
Out[22]:TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
                        '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
                       dtype='timedelta64[ns]', freq='H')
```

이 시퀀스는 모두 Pandas주기 코드를 이해해야 한다.

주기(Frequencies)와 오프셋(Offsets)

이 Pandas시계열 도구는 주기나 날자 오프셋 개념을 기반으로 한다. 앞으로 본 D(day)와 H(hour)코드처럼 그러한 코드를 사용해 원하는 주기 간격을 지정할 수 있다.

Code	Description	Code	Description
D	달력상 일	B	영업일
W	주		
M	월말	BM	영업일 기준 월말
Q	분기 말	BQ	영업일 기준 분기 말
A	년말	BA	영업일 기준 년말
H	시간	BH	영업일 시간
T	분		
S	초		
L	밀리초		
U	마이크로초		
N	나노초		

월, 분기, 연 단위의 주기는 모두 지정한 기간의 종료 시점을 표시한다. 이 표시에 접미사 5를 추가하면 종료가 아니라 시작 시점으로 표시된다.

Code	Description	Code	Description
MS	월초	BMS	영업일 기준 분기 초
QS	분기초	BQS	영업일 기준 분기 초
AS	년초	BAS	영업일 기준 년 초

이 밖에도 세 글자로 구성된 월 코드를 접미사로 추가해 분기나 년 코드를 표시하는데 사용되는 월을 바꿀 수 있다.
- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

같은 방법으로 세글자로 구성된 요일 코드를 추가해서 주를 나누는 분할 점을 수정할 수 있다.
- W-SUN, W-MON, W-TUE, W-WED, etc.

게다가 이 코드를 숫자와 결합해 다른 주기를 지정할 수도 있다.
2시간 30분 간격의 주기를 지정하려면 시간(H)과 분(T)코드를 다음과 같이 결합하면 된다.

```
In [23]:pd.timedelta_range(0, periods=9, freq="2H30T")
Out[23]:TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',
                        '12:30:00', '15:00:00', '17:30:00', '20:00:00'],
                       dtype='timedelta64[ns]', freq='150T')
```

여기에 사용된 짧은 코드는 모두 pd.tseries.offsets모듈에서 찾아볼 수 있는 Pandas시계열 오프셋의 특정 인스턴스를 가르킨다.
영업일 오프셋을 바로 만들 수 있다.

```
In [24]:from pandas.tseries.offsets import BDay
        pd.date_range('2015-07-01', periods=5, freq=BDay())
Out[24]:DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
                       '2015-07-07'],
                      dtype='datetime64[ns]', freq='B')
```

주기와 오프셋의 사용법에 대해 더 알아보고 싶으면 Pandas온라인 문서의 "DateOffset 객체" 절을 참고한다.

리샘플링, 시프팅, 윈도잉

데이터를 직관적으로 구성하고 접근하기 위해 날짜와 시간을 인덱스로 사용하는 능력은 Pandas시계열 도구의 중요한 부분이다. 일반적으로 인덱스를 가진 데이터의 이점(연산하는 동안 자동 정렬, 직관적인 데이터 슬라이싱 및 접근 등)과 함께 Pandas는 몇 가지 추가적인 시계열 전용 연산을 제공한다.

연산 몇 가지에 대해 주가 데이터를 예로 들어 살펴보겠다. Pandas는 주로 금융 환경에서 개발됐기 때문에 몇몇 금융 데이터에 특화된 전용 도구를 포함하고 있다. 예를 들면 pandas-datareader패키지(conda install pandas-datareadr를 통해 설치할 수 있다)는 야후 파이낸스,구글 파이낸스 등을 포함한 다양한 소스로부터 금융 데이터를 임포트하는데 능숙하다.

yahoo의 종가 이력 적재

```
In [25]:from pandas_datareader import data
        goog = data.DataReader('GOOG', start='2004', end='2016',
                               data_source='yahoo')
        goog.head()
```

Out[25]:

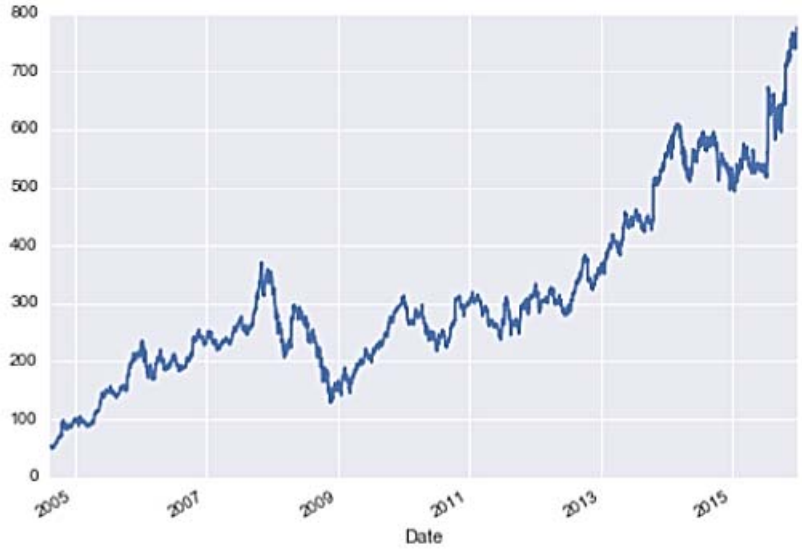
Date	Open	High	Low	Close	Volume
2004-08-19	49.96	51.98	47.93	50.12	NaN
2004-08-20	50.69	54.49	50.20	54.10	NaN
2004-08-23	55.32	56.68	54.47	54.65	NaN
2004-08-24	55.56	55.74	51.73	52.38	NaN
2004-08-25	52.43	53.95	51.89	52.95	NaN

문제를 간단하게 만들기 위해 종가만 사용한다.

```
In [26]:goog = goog['Close']
```

이 데이터를 일반적인 표준 Mathplotlib설정 구문 다음에 plot()메서드를 사용해 시각화할 수 있다.

```
In [27]:%matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()
In [28]:goog.plot();
```

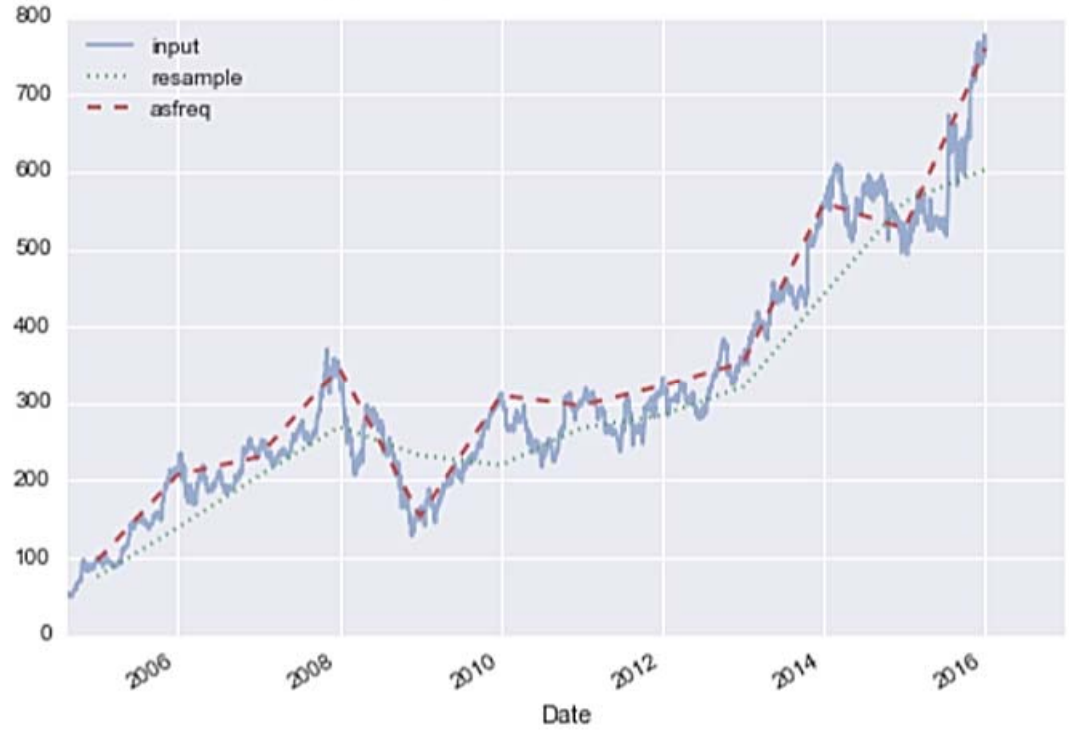


리샘플링 및 주기 변경

시계열 데이터에서 일반적으로 필요한 작업은 더 높거나 낮은 주기로 표본을 다시 추출(resampling)하는 것이다. 이 작업은 resample()메서드를 사용하거나 훨씬 더 간단한 asfreg()메서드를 사용해 수행할 수 있다. 이 두 메서드의 주요 차이점은 resample()은 기본적으로 데이터를 집계하지만 asfreg()는 기본적으로 데이터를 선택한다는 것이다.

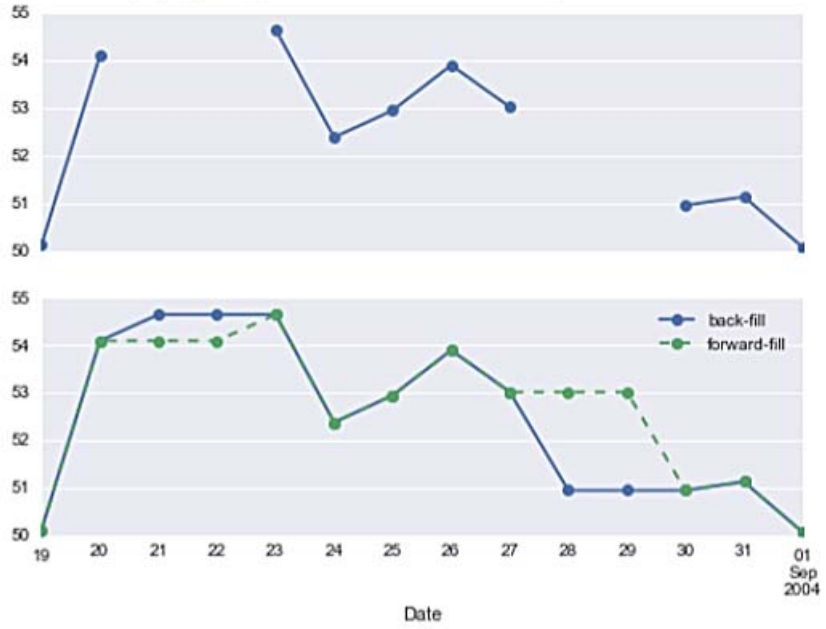
야후 종가를 살펴보면서 데이터를 다운 샘플링(down-sampling)할 때 이 두개의 메서드가 반환하는 값을 비교해 보자. 여기서는 영업일 기준 연말 데이터를 리샘플링할 것이다.

```
In [29]: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');
```



차이점은 각 점에서 resample은 전년도 평균을 보여주지만, asfreg는 연말 주기를 보여준다는 것이다.
업샘플링(upsampling)의 경우 resample()과 asfreg()가 대체로 유사하지만, resample메서드에서 더 많은 옵션을 사용할 수 있다. 이 경우에 두 메서드의 기본값은 업샘플링된 점을 빈 값으로 두는 것이다. 즉 NA값으로 채운다. 앞에서 본 pd.fillna()함수와 마찬가지로 asfreg()는 값을 어떤 방식으로 채울 것인지를 지정하기 위해 method인수를 받는다. 여기서는 영업일 데이터를 일별 주기로(주말 포함) 리샘플링할 것이다.

```
In [30]: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(["back-fill", "forward-fill"]);
```



위 그림은 기본 그래프로, 영업일이 아닌 날은 NA값으로 두기 때문에 그래프 상에 표시되지 않는다. 아래 그림은 그 틈을 채우기 위한 두 가지 전략인 순방향 채우기와 역방향 채우기의 차이를 보여준다.

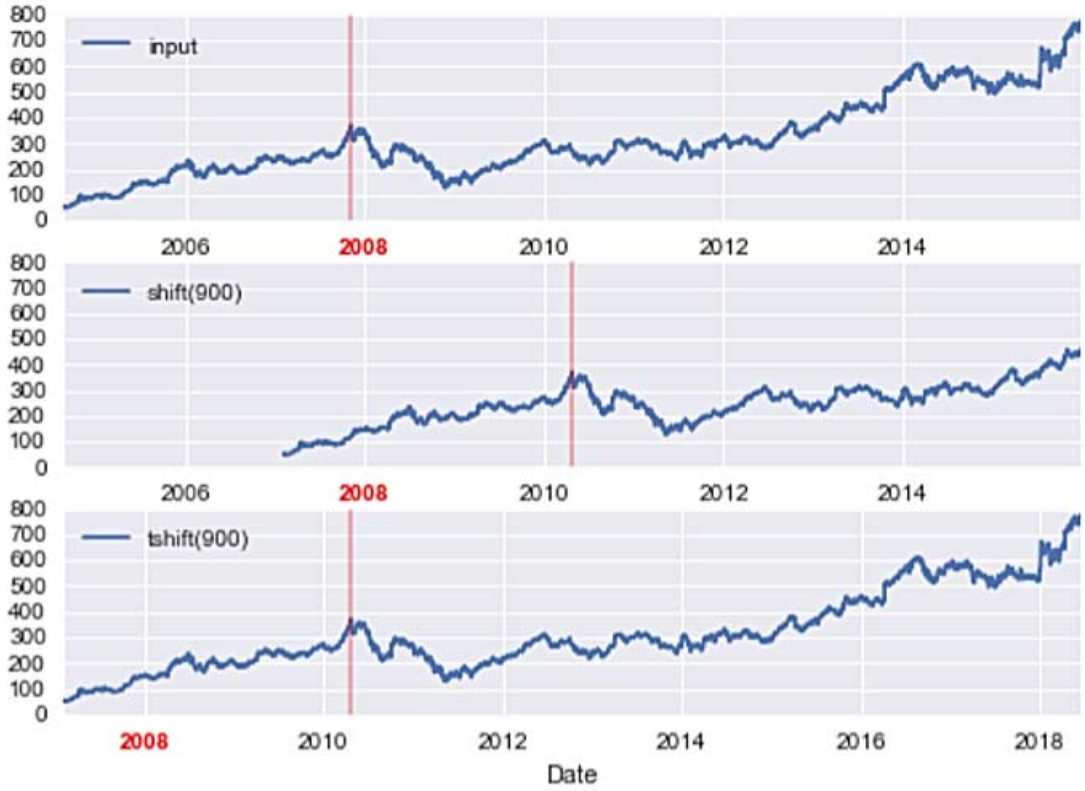
(시간 이동)Time-shifts¶

또 다른 일반적인 시계열 전용 연산은 시간에 따라 데이터를 이동시키는 것이다. Pandas에는 이를 계산하기 위해 두 가지 밀접하게 관련된 메서드 shift()와 tshift()가 있다. 요컨대 이 두 메서드 사이의 차이는 shift()는 데이터를 이동시키는 반면 tshift()는 인덱스를 이동시킨다는데 있다. ;

900일 단위로 shift()와 tshift()를 수행할 것이다.

```
In [31]: import pandas as pd
         fig, ax = plt.subplots(3, sharey=True)
         # 데이터에 주기를적용
         goog = goog.asfreq('D', method='pad')
         goog.plot(ax=ax[0])
         goog.shift(900).plot(ax=ax[1])
         goog.tshift(900).plot(ax=ax[2])

         #범례와 주석
         local_max = pd.to_datetime('2007-11-05')
         offset = pd.Timedelta(900, 'D')
```

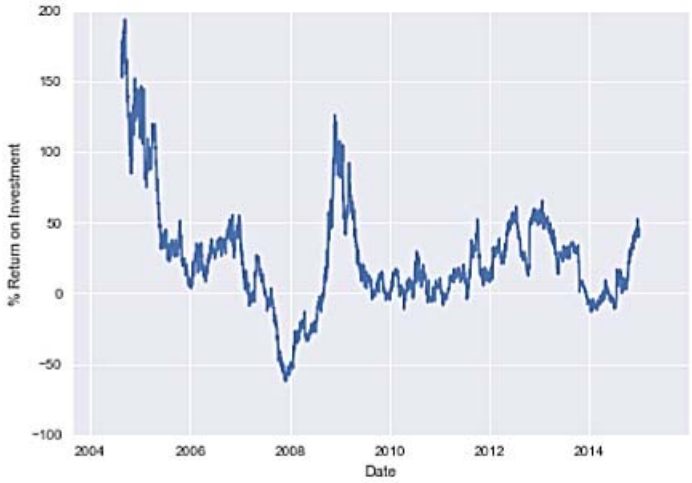
```
ax[0].legend(['input'], loc=2)
ax[0].get_xticklabels()[2].set(weight='heavy', color='red')
ax[0].axvline(local_max, alpha=0.3, color='red')

ax[1].legend(['shift(900)'], loc=2)
ax[1].get_xticklabels()[2].set(weight='heavy', color='red')
ax[1].axvline(local_max + offset, alpha=0.3, color='red')

ax[2].legend(['tshift(900)'], loc=2)
ax[2].get_xticklabels()[1].set(weight='heavy', color='red')
ax[2].axvline(local_max + offset, alpha=0.3, color='red');
```



여기서 shift(900)은 데이터를 900일 이동시켜서 그래프 끝의 일부를 밀어내고 그 반대쪽은 NA값으로 두는 반면, tshift(900)은 인덱스값을 900일 만큼 이동시킨다.:

이러한 유형의 이동 작업은 시간상의 차이를 계산할 때 일반적으로 사용된다. 다음 예제는 이동된 값을 사용해 데이터세트의 기간 동안 야후 주가의 연간 투자 대비 효과를 계산한 것이다.

In [32]:
```
ROI = 100 * (goog.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Return on Investment');
```



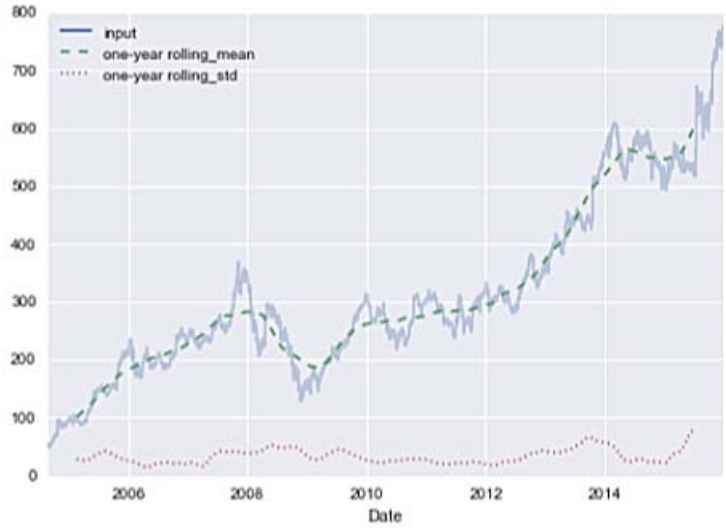
이 그래프는 구글 주식의 전반적 추세를 보여준다. 지금까지 구글에 투자하기가장 좋았던 시점은 구글의 IPO바로 다음과 2009년 불황기 때였다.

(롤링 윈도우)Rolling windows

Pandas에 구현된 시계열 전용 연산의 세번째 유형으로는 롤링 통계가 있다. 이 롤링 통계는 Series와 DataFrame객체의 rolling()속성을 통해 수행할 수 있는데, 이 속성은 groupby연산에서 본 것과 유사한 뷰를 반환한다. 이 롤링 뷰에서는 여러 집계 연산을 기본으로 사용할 수 있다.

야후 주가의 1년 중심 롤링 평균과 표준편차를 구한 것이다

```
In [33]:rolling = goog.rolling(365, center=True)
        data = pd.DataFrame({'input': goog,
                             'one-year rolling_mean': rolling.mean(),
                             'one-year rolling_std': rolling.std()})
        ax = data.plot(style=['-', '--', ':'])
        ax.lines[0].set_alpha(0.3)
```



Groupby 연산과 마찬가지로 사용자 정의 롤링 계산을 위해 aggregate()와 apply()메서드를 사용할 수 있다.

추가 학습 자료¶

Pandas에서 제공하는 시계열 도구의 가장 기본적인 기능 중 일부만 요약했다. 온라인 문서인 '시계열 / 날짜 ("Time Series/Date" section)절'을 참고하면 된다.

예제: 시애틀 자전거 수 시각화¶

시계열 데이터로 작업하는 좀 더 복잡한 예제로 시애틀의 프로몬트 다리를 통행하는 자전거 수를 살펴보자. 이 데이터는 2012년 후반에 설치된 다리의 동쪽 서쪽 보도에 유도 센서를 가지고 있는 자동 자전거 제수기에 의해 집계된다.시간별자전거 수는다은 url에서 다운로드 할 수 있다.

- https://data.seattle.gov

해당 데이터세트는 다음 URL을 통해 바로 확인할 수 있다:

- https://data.seattle.gov/Transportation/Fremont-Bridge-Hourly-Bicycle-counts-by-Month-Octo/65db-xm6k

2017년 현재 CVS파일은 다음과 같이 내려 받을 수 있다.

In [34]:
```
# !curl -o FremontBridge.csv https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD
```

이 데이터세트를 내려받고 나면 Pandas를 사용해 CVS를 DataFrame으로 읽어 들일 수 있다. 인덱스로 Data를 사용하고 이 날짜를 자동 분석하는 것으로 지정할 것이다.

In [35]:
```
data = pd.read_csv('data/FremontBridge.csv', index_col='Date', parse_dates=True)
data.head()
```

Out[35]:

Date	Fremont Bridge West Sidewalk	Fremont Bridge East Sidewalk
2012-10-03 00:00:00	4.0	9.0
2012-10-03 01:00:00	4.0	6.0
2012-10-03 02:00:00	1.0	1.0
2012-10-03 03:00:00	2.0	3.0
2012-10-03 04:00:00	6.0	1.0

편의상 이 데이터세트의 열 이름을 단축하자.

In [36]:
```
data.columns = ['Total', 'East', 'West']
```

이제 이 데이터의 요약 통계를 살펴보자.

In [37]:
```
data.dropna().describe()
```

Out[37]:

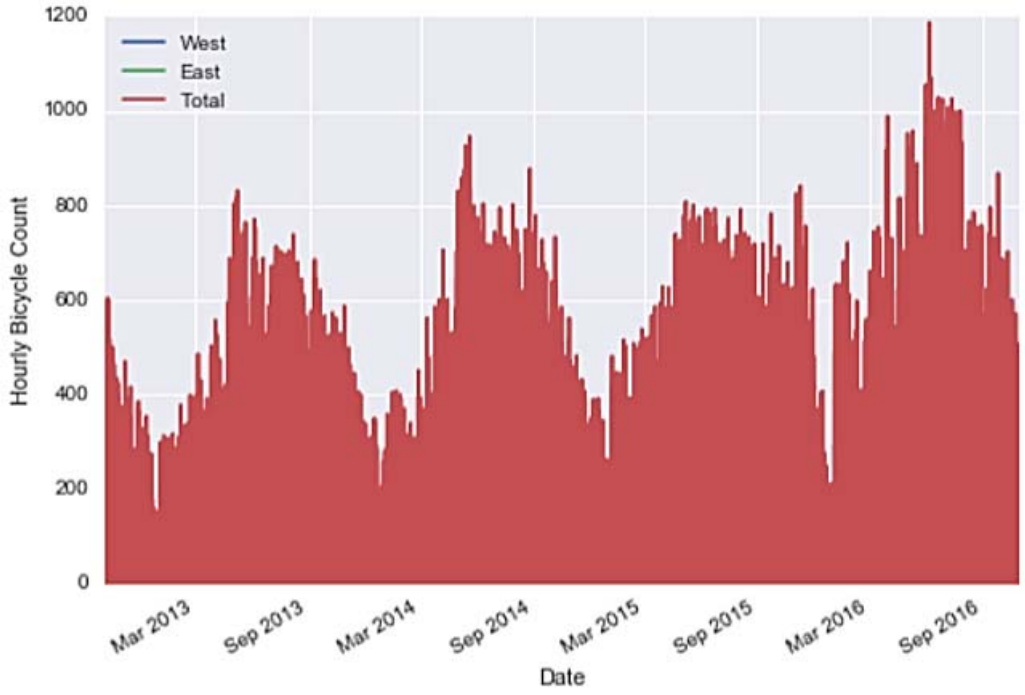
	West	East	Total
count	35752.000000	35752.000000	35752.000000
mean	61.470267	54.410774	115.881042
Std	82.588484	77.659796	145.392385
Min	0.000000	0.000000	0.000000

25%	8.000000	7.000000	16.000000
50%	33.000000	28.000000	65.000000
75%	79.000000	67.000000	151.000000
Max	825.000000	717.000000	1186.000000

데이터 시각화하기

이 데이터세트를 시각화해 보면 몇 가지 통찰력을 얻을 수 있다. 우선 원시 데이터를 그래프로 나타내보자.:

```
In [38]:%matplotlib inline
        import seaborn; seaborn.set()
In [39]:data.plot()
        plt.ylabel('Hourly Bicycle Count');
```



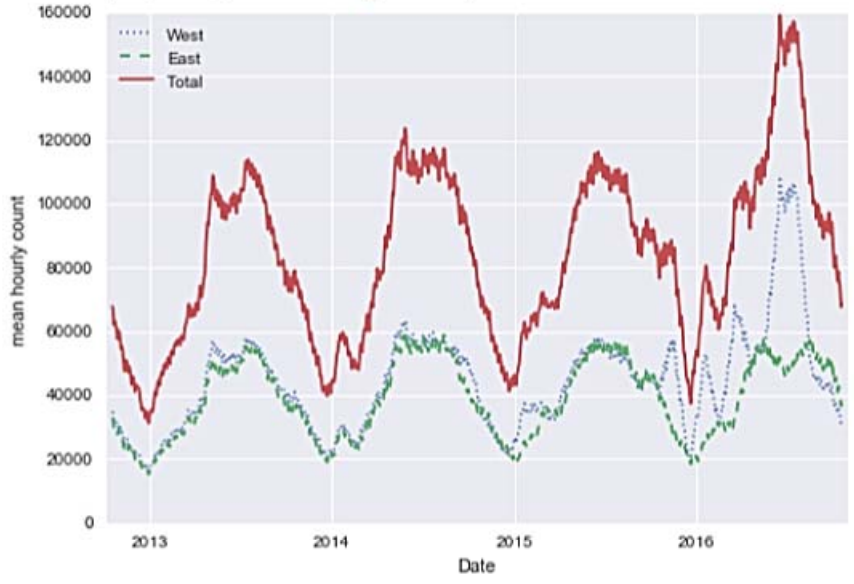
25000개까지의 시간별 표본은 너무 밀집돼 있어서 이해하기가 어렵다. 좀 더 성긴 그리드에 데이터를 리샘플링하면 더 많은 통찰력을 얻을 수 있다.

주 단위로 샘플링해 보자.

```
In [40]:weekly = data.resample('W').sum()
        weekly.plot(style=[':', '--', '-'])
        plt.ylabel('Weekly bicycle count');
```

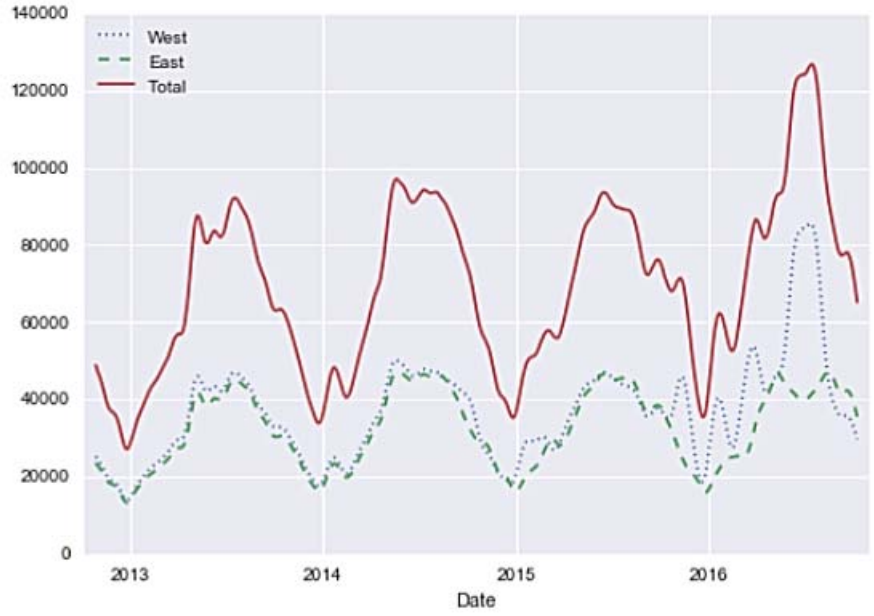
이코드는 몇가지 흥미로운 계절성을 보여준다. 예상했듯이 사람들은 겨울보다 여름에 자전거를 더 많이 타며 특정 계절에는 자전거 사용횟수가 주마다 달라진다.

```
In [41]:daily = data.resample('D').sum()
        daily.rolling(30, center=True).sum().plot(style=[':', '--', '-'])
        plt.ylabel('mean hourly count');
```



결과가 들쑥날쑥한 것은 기간을 잘라서 표기했기 때문이다. 가우스 윈도우(Gausian window)같은 윈도우 함수를 사용해 롤링 평균을 부드럽게 표현할 수 있다. 다음 코드는 윈도우 폭(50일로 선택)과 윈도우 내 가우스 폭(10일로 선택)을 모두 지정한다

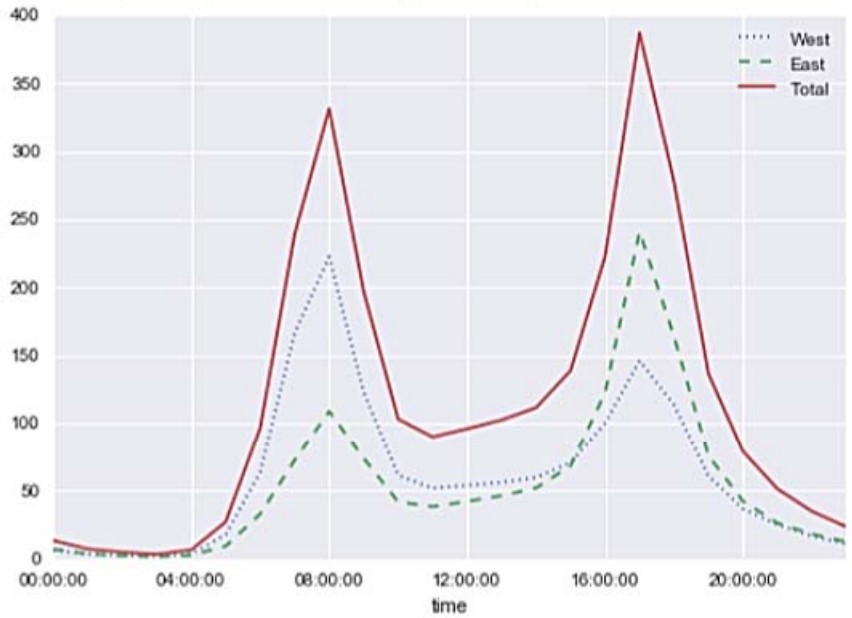
```
In [42]:daily.rolling(50, center=True,
                win_type='gaussian').sum(std=10).plot(style=[':', '--', '-']);
```



데이터 파헤쳐 보기

위 그래프의 평활된 데이터 뷰는 데이터의 일반적인 추세를 살펴보는 데는 유용하지만 많은 흥미로운 구조를 보여주지 않는다. 예를 들면 , 하루의 사간대를 기준으로 한 함수로 평균 통행량을 보고 싶다고 하자. 이 작업은 "집계와 분류"에서 소개한 groupby기능을 사용해 수행할 수 있다.

```
In [43]:by_time = data.groupby(data.index.time).mean()
        hourly_ticks = 4 * 60 * 60 * np.arange(6)
        by_time.plot(xticks=hourly_ticks, style=[':', '--', '-']);
```

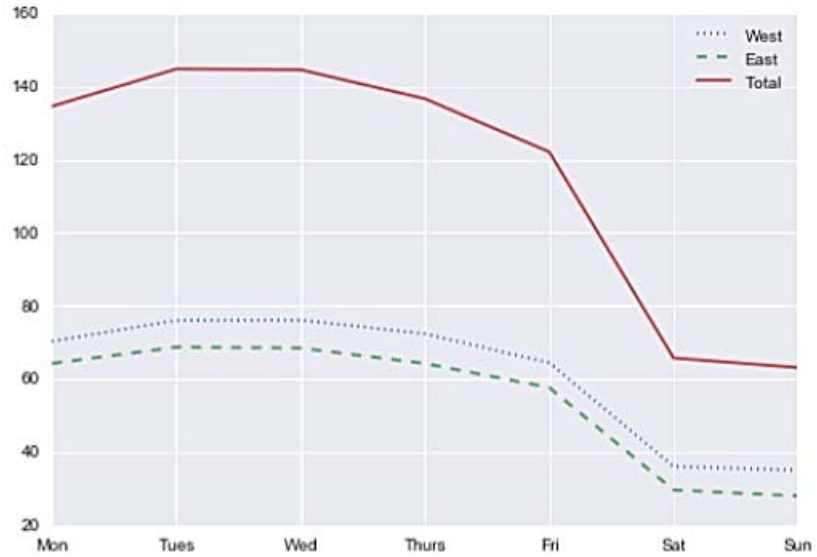


시간대별 통행량은 아침 8시와 저녁 5시 무렵에 정점을 이루는 강한 양봉 분포를 보인다. 이는 다리를 건너는 출근 통행량인것으로 보인다. 이 추측의 또 다른 근거로 서쪽 보도(시애틀 도심으로 이동할 때 주로 사용됨)의 교통량이 아침에 정점을 이루고 동쪽 보도(시애틀 도심에서 벗어날 때 주로 사용됨)가 저녁에 정점을 찍는다는 사실을 들 수 있다.

요일에 따라 통행량이 어떻게 변하는지 궁금할 수도 있다. 이 문제 역시 간단한 groupby로 답을 구할 수 있다.

```
In [44]:by_weekday = data.groupby(data.index.dayofweek).mean()
```

```
by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
by_weekday.plot(style=[':', '--', '-']);
```



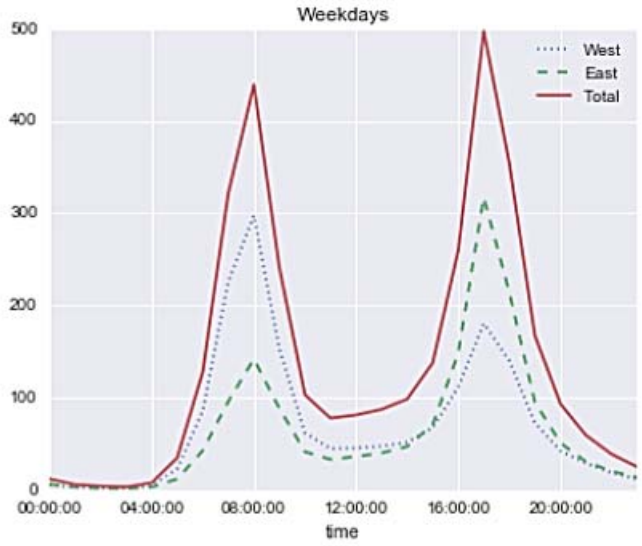
여기서 주중과 주말 총합 사이에 차이가 뚜렷하게 드러난다. 월요일부터 금요일까지 평균 통행량이 토요일과 일요일의 평균 통행량의 두 배 정도다.

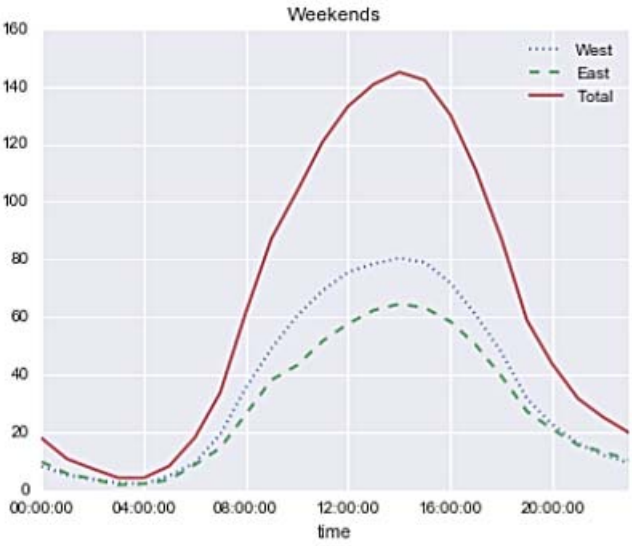
이 사실을 염두에 두고 복합적인 groupby를 사용해 주중과 주말의 시간대별 추이를 살펴본다. 먼저 데이터를 주말을 표시하는 플래그와 시간대별로 분류하자.

```
In [45]:weekend = np.where(data.index.weekday < 5, 'Weekday', 'Weekend')
        by_time = data.groupby([weekend, data.index.time]).mean()
```

"다중 서브플롯"에서 설명한 Mathplotlib 구의 일부를 사용해 두 그래프를 나란히 그려보자.

```
In [46]:import matplotlib.pyplot as plt
        fig, ax = plt.subplots(1, 2, figsize=(14, 5))
        by_time.loc['Weekday'].plot(ax=ax[0], title='Weekdays',
                                    xticks=hourly_ticks, style=[':', '--', '-'])
        by_time.loc['Weekend'].plot(ax=ax[1], title='Weekends',
                                    xticks=hourly_ticks, style=[':', '--', '-']);
```





그래프가 주중에 양봉 형태의 출퇴근 패턴을 보이고 주말에는 단봉의 여가를 즐기는 패턴을 보인다. 이 데이터를 더 자세히 분석해 사람들의 출퇴근 패턴에 영향을 미치는 날씨와 온도, 연중 시기 등 기타 요인의 효과를 알아보면 흥미로울 것이다. 더 많은 내용은 이 데이터세트의 하위 집합을 사용한 저자의 블로그 포스트 '시애틀의 자전거 이용자가 정말 늘어나고 있는 걸까?