11. 시계열

시계열 데이터는 금융, 경제, 생태학, 신경과학, 물리학 등 여러 다양한 분야에서 사용되는 매우 중요한 구조화된 데이터다. 시간상의 여러 지점을 관측하거나 측정할 수 있는 모든 것이 시계열이다. 대부분의 시계열은 **고정 빈도**로 표현되는데 데이터가 존재하는 지검이 15초마다, 5분 마다, 한 달에 한 번 같은 특정 규칙에 따라 고정 간격을 가진다. 시계열은 또한 고정된 단위나 시간 혹은 단위들 간의 간격으로 존재하지 않고 **불규칙적인** 모습으로 표현될 수도 있다. 어떻게 시계열 데이터를 표시하고 참조할지는 애플리케이션에 의존적이며 다음 중 한 유형일 수 있다.

- 시간 내에서 특정 순간의 타임스탬프
- 2007년 1월 이나 2010년 전체 같은 고정된 기간
- 시작과 끝 타임스탬프로 표시되는 시간 간격, 기간은 시간 간격의 특수한 경우로 생각할 수 있다.
- 실험 혹은 경과 시간, 각 타임스탬프는 특정 시작 시간에 상대적인 시간의 측정값이다. 특정 시작 시간에 상대적인 시간의 측정값이다. (예 : 쿠키를 오븐에 넣은 시점부터 매 초가 지날 때마다 쿠키 반죽의 지름)

실험의 시작 시점부터의 경과 시간이 정수나 부동소수점으로 표현되는 경우 실험 시계열에도 해당 기술들을 적용할 수 있지만 이 장에서는 위에서 소개한 시계열 데이터의 처음 3가지 종류에 대해 주로 알아볼 것이다. 가장 단순하고 널리 사용되는 시계열의 종류는 타임스탬프로 색인된 데이터다.

pandas는 표준 시계열 도구와 데이터 알고리즘을 제공한다. 이를 통해 대량의 시계열 데이털르 효과적으로 다룰 수 있으며 쉽게 나누고, 집계하고, 불규칙적이며 고정된 빈도를 갖는 시계열을 리샘플링할 수 있다. 눈치 챘겠지만 대부분의 도구는 금융이나 경제 관련 애플리케이션에서 특히 유용하다. 하지만 서버 로그 데이터를 분석하는 데도 사용할 수 있다.

11.1 날짜, 시간, 자료형, 도구

파이썬 표준 라이브러리는 날짜와 시간을 위한 자료형과 달력 관련 기능을 제공하는 자료형이 존재한다. datetime, time 그리고 calendar 모듈은 처음 공부하기에 좋은 주제다. datetime, datetime 형이나 단순한 datetime 이 널리 사용되고 있다.

```
In [76]: from datetime import datetime

In [78]: now = datetime.now()

In [79]: now
Out[79]: datetime.datetime(2020, 5, 20, 12, 0, 29, 257977)

In [80]: now.year, now.month, now.day
Out[80]: (2020, 5, 20)
```

datetime은 날짜와 시간을 모두 저장하며 마이크로초까지 지원한다. datetime.timedelta는 두 datetime 객체 간의 시간적인 차이를 표현할 수 있다.

```
In [81]: delta = datetime(2011,1,7)-datetime(2008,6,24,8,15)

In [82]: delta
Out[82]: datetime.timedelta(days=926, seconds=56700)

In [83]: delta.days
Out[83]: 926

In [84]: delta.seconds
Out[84]: 56700
```

timedelta를 더하거나 빼면 그만큼의 시간이 datetime 객체에 적용되어 새로운 객체를 만들 수 있다.

```
In [86]: start = datetime(2011,1,7)

In [87]: start + timedelta(12)
Out[87]: datetime.datetime(2011, 1, 19, 0, 0)

In [88]: start - 2 * timedelta(12)
Out[88]: datetime.datetime(2010, 12, 14, 0, 0)
```

다음 표에 datetime 모듈의 자료형을 정리해뒀다. 이 장에서 주로 다루는 내용은 pandas의 자료형과 고수준의 시계열을 다루는 방법이며, 실제 파이썬을 사용하면서 다양한 곳에서 datetime 기반의 자료형을 마주치게 되리라는 점은 의심할 여지가 없다.

datetime 모듈의 자료형

자료형	설명
date	그레고리안 달력을 사용해서 날짜 (연,월,일)를 저장한다.
time	하루의 시간을 시,분,초 마이크로초 단위로 저장한다.
datetime	날짜와 시간을 저장한다.
timedelta	두 datetime 값 간의 차이(일,초, 마이크로초)를 표현한다.
tzinfo	지역시간대를 저장하기 위한 기본 자료형

11.1.1 문자열을 datetime으로 변환하기

datetime 객체와 나중에 소개할 pandas의 Timestamp 객체는 str 메서드나 strftime 메서드에 포맷 규칙을 넘겨서 문자열로 나타낼 수 있다.

```
In [89]: stamp = datetime(2011,1,3)

In [90]: str(stamp)
Out[90]: '2011-01-03 00:00:00'

In [91]: stamp.strftime('%Y-%m-%d')
Out[91]: '2011-01-03'
```

포맷 코드를 모두 정리했다.

포맷	설명
%Y	4자리 연도
%y	2자리 연도
%m	2자리 월[01,12]
%d	2자리 일 [01,31]
%H	시간(24시간 형식)[00,23]
%I	시간(12시간 형식)[01,12]
%M	2자리 분 [00,59]
%S	초[00,61](60,61은 윤초)
%w	정수로 나타낸 요일[0(일요일), 6]
%U	연중 주차 [00,53]. 일요일을 그 주의 첫 번째 날로 간주하며, 그 해에서 첫 번째 일요일 앞에 있는 날은 0주차가 된다.
%W	연중 주차 [00,53]. 월요일을 그 주의 첫 번째 날로 간주하며, 그 해에서 첫 번째 일요일 앞에 있는 날은 0주차가 된다.
%z	UTC 시간대 오프셋을 +HHMM 또는 -HHMM으로 표현한다. 만약 시간대를 신경 쓰지 않는다면 비워둔다.
%F	%Y-%m-%d 형식에 대한 축약(예 : 2012-4-18)
%D	%m/%d/%y 형식에 대한 축약(예: 04/18/12)

이 포맷 코드는 datetime.strptime을 사용해서 문자열을 날짜로 변환할 때 사용할 수 있다.

```
In [94]: value = '2011-01-03'

In [95]: datetime.strptime(value, '%Y-%m-%d')
Out[95]: datetime.datetime(2011, 1, 3, 0, 0)

In [96]: datestrs = ['7/6/2011','8/6/2011']

In [97]: [datetime.strptime(x,'%m/%d/%Y') for x in datestrs]
Out[97]: [datetime.datetime(2011, 7, 6, 0, 0),
          datetime.datetime(2011, 8, 6, 0, 0)]
```

datetime.strptime은 알려진 형식의 날짜를 파싱하는 최적의 방법이다. 하지만 매번 포맷 규칙을 써야 하는 것은 귀찮은 일이다. 특히 흔히 쓰는 날짜 형식에 대해서는 더 그렇다. 이 경우에는 서드파티 패키지인 dateutil에 포함된 parser.parse 메서드를 사용하면 된다.

```
In [98]: from dateutil.parser import parse

In [99]: parse('2011-01-03')
Out[99]: datetime.datetime(2011, 1, 3, 0, 0)
```

dateutil은 거의 대부분의 사람이 인지하는 날짜 표현 방식을 파싱할 수 있다.

```
In [100]: parse('Jan 31,1997 10:45PM')
Out[100]: datetime.datetime(2020, 1, 31, 22, 45)
```

국제 로케일의 경우 날짜가 월 앞에 오는 경우가 많다. 이때 dayfirst 옵션을 써준다.

```
In [101]: parse('6/12/2011', dayfirst = True)
Out[101]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas는 일반적으로 DateFrame의 컬럼이나 축 색인으로 날짜가 담긴 배열을 사용한다.

to_datetime 메서드는 많은 종류의 날짜 표현을 처리한다.

```
In [102]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [103]: pd.to_datetime(datestrs)
Out[103]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'],
dtype='datetime64[ns]', freq=None)
```

또한 누락된 값(None, 빈 문자열 등) 으로 간주되어야 할 값도 처리해준다.

```
In [104]: idx = pd.to_datetime(datestrs + [None])

In [105]: idx
Out[105]: DatetimeIndex(['2011-07-06 12:00:00',
'2011-08-06 00:00:00', 'NaT'], dtype='datetime64[ns]', freq=None)

In [106]: idx[2]
Out[106]: NaT

In [107]: pd.isnull(idx)
Out[107]: array([False, False,  True])
```

CAUTION_ dateutil.parser는 매우 요용하지만 완벽한 도구는 아니다. 날짜로 인식하지 않길 바라는 문자열을 날짜로 인식하기도 하는데 '42'를 2042년으로 해석하기도 한다.

포맷	설명
%a	축약된 요일 이름
%A	요일 이름
%b	축약된 월 이름
%B	월 이름
%c	전체 날짜와 시간('Tue 01 May 2012 04:20:57 PM')
%p	해당 로케일에서 AM,PM에 대응되는 이름
%x	로케일에 맞는 날짜 형식
%X	로케일에 맞는 시간 형식

11.2 시계열 기초

pandas에서 찾아볼 수 있는 가장 기본적인 시계열 객체의 종류는 파이썬 문자열이나 datetime객체로 표현되는 타임스탬프로 색인된 Series다.

```
In [76]: from datetime import datetime

In [108]: 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)]


In [109]: ts = pd.Series(np.random.randn(6),index = dates)

In [110]: ts
Out[110]:
2011-01-02   -1.699012
2011-01-05   -0.215649
2011-01-07    2.473118
2011-01-08    0.259266
2011-01-10   -1.022399
2011-01-12    1.284342
dtype: float64
```

내부적으로 보면 이들 datetime 객체는 DatetimeIndex에 들어 있으며 ts 변수의 타입은 TimeSeries다.

```
In [111]: ts.index
Out[111]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

다른 Series와 마찬가지로 서로 다르게 색인된 시계열 객체 간의 산술 연산은 자동으로 날짜에 맞춰진다.

```
In [112]: ts
Out[112]:
2011-01-02   -1.699012
2011-01-05   -0.215649
2011-01-07    2.473118
2011-01-08    0.259266
2011-01-10   -1.022399
2011-01-12    1.284342
dtype: float64

In [113]: ts[::2]
Out[113]:
2011-01-02   -1.699012
2011-01-07    2.473118
2011-01-10   -1.022399
dtype: float64

In [114]: ts + ts[::2]
Out[114]:
2011-01-02   -3.398024
2011-01-05         NaN
2011-01-07    4.946236
2011-01-08         NaN
2011-01-10   -2.044798
```

```
2011-01-12         NaN
dtype: float64
```

ts[::2]에서 매 두 번째 항목을 선택한다. pandas는 Numpy의 datetime64 자료형을 사용해서 나노초의 정밀도를 가지는 타임스탬프를 저장한다.

```
In [115]: ts.index.dtype
Out[115]: dtype('<M8[ns]')
```

DatetimeIndex의 스칼라값은 pandas의 Timestamp 객체다.

```
In [116]: stamp = ts.index[0]

In [117]: stamp
Out[117]: Timestamp('2011-01-02 00:00:00')
```

Timestamp는 datetime 객체를 사용하는 어떤 곳에도 대체 사용이 가능하다. 게다가 가능하다면 빈도에 관한 정보도 저장하며 시간대 변환을 하는 방법과 다른 종류의 조작을 하는 방법도 포함하고 있다.

11.2.1 색인, 선택, 부분 선택

시계열은 라벨에 기반해서 데이터를 선택하고 인덱싱할 때 pandas.Series와 동일하게 동작한다.

```
In [116]: stamp = ts.index[0]

In [117]: stamp
Out[117]: Timestamp('2011-01-02 00:00:00')

In [118]: stamp = ts.index[2]

In [119]: ts[stamp]
Out[119]: 2.473118154553548

In [120]: stamp
Out[120]: Timestamp('2011-01-07 00:00:00')

In [121]: ts[ts.index[2]]
Out[121]: 2.473118154553548
```

해석할 수 있는 날짜를 문자열로 넘겨서 편리하게 사용할 수 있다.

```
In [125]: ts['20110110']
Out[125]: -1.0223991446761098

In [126]: ts['1/10/2011']
Out[126]: -1.0223991446761098
```

긴 시계열에서는 연을 넘기거나 연, 월 만 넘겨서 데이터의 일부 구간만 선택할 수도 있다.

```
In [127]: longer_ts = pd.Series(np.random.randn(1000),
     ...:                        index = pd.date_range('1/1/2000'
     ...:                        ,periods=1000))

In [128]: longer_ts
```

```
Out[128]:
2000-01-01    1.288661
2000-01-02   -1.174924
2000-01-03   -0.067708
2000-01-04    0.164763
2000-01-05    1.062469
                ...
2002-09-22   -0.288743
2002-09-23    0.786957
2002-09-24   -0.696627
2002-09-25   -1.816090
2002-09-26   -1.137660
Freq: D, Length: 1000, dtype: float64
```

여기서 문자열 '2001'은 연도로 해석되어 해당 기간의 데이터를 선택한다. 월에 대해서도 마찬가지로 선택할 수 있다.

```
In [137]: longer_ts['2001/05'].head()
Out[137]:
2001-05-01    0.209742
2001-05-02    2.193308
2001-05-03    0.144784
2001-05-04   -0.639322
2001-05-05    1.393109
Freq: D, dtype: float64
```

datetime 객체로 데이터를 잘라내는 작업은 일반적인 Series와 동일한 방식으로 할 수 있다.

```
In [138]: ts[datetime(2011,1,7):]
Out[138]:
2011-01-07    2.473118
2011-01-08    0.259266
2011-01-10   -1.022399
2011-01-12    1.284342
dtype: float64
```

대부분의 시계열 데이터는 연대순으로 정렬되기 때문에 범위를 지정하기 위해 시계열에 포함하지 않고 타임스탬프를 이용해서 Series를 나눌 수 있다.

```
In [139]: ts
Out[139]:
2011-01-02   -1.699012
2011-01-05   -0.215649
2011-01-07    2.473118
2011-01-08    0.259266
2011-01-10   -1.022399
2011-01-12    1.284342
dtype: float64

In [140]: ts['2011/01/06':'2011/01/12']
Out[140]:
2011-01-07    2.473118
2011-01-08    0.259266
2011-01-10   -1.022399
2011-01-12    1.284342
```

```
dtype: float64
```

앞서와 같이 날짜 문자열이나 datetime 혹은 타임스탬프로 넘길 수 있다. 이런 방식으로 데이터를 나누면 NumPy 배열을 나누는 것처럼 원본 시계열에 대한 뷰를 생성한다는 사실을 기억하자.

즉, 데이터 복사가 발생하지 않고 슬라이스에 대한 변경이 원본 데이터에도 반영된다.

이와 동일한 인스턴스 메서드로 truncate가 있는데, 이 메서드는 TimeSeries를 두 개의 날짜로 나눈다.

```
In [141]: ts.truncate(after = '1/9/2011')
Out[141]:
2011-01-02   -1.699012
2011-01-05   -0.215649
2011-01-07    2.473118
2011-01-08    0.259266
dtype: float64
```

위 방식은 DataFrame에서도 동일하게 적용되며 로우에 인덱싱된다.

```
In [142]: dates = pd.date_range('1/1/2000', periods = 100,
     ...:                        freq = 'W-WED')

In [144]: long_df = pd.DataFrame(np.random.randn(100,4),
     ...:                        index = dates,
     ...:                        columns = ['Colorado','Texas','New York'
     ...:                        , 'Ohio'])

In [145]: long_df.loc['5-2001']
Out[145]:
            Colorado     Texas  New York      Ohio
2001-05-02  1.366816  0.171642 -0.943959  0.337466
2001-05-09 -0.905977 -0.058252  0.767248  0.735015
2001-05-16 -2.244084  0.292371  0.025099 -0.099385
2001-05-23 -2.608691  0.311730  0.213023  0.222122
2001-05-30 -0.491371  0.397196 -2.001426  0.344459
```

11.2.2 중복된 색인을 갖는 시계열

어떤 어플리케이션에서는 여러 데이터가 특정 타임스탬프에 몰려 있는 것을 발견할 수 있다.

```
In [146]: dates = pd.DatetimeIndex(['1/1/2000','1/2/2000','1/2/2000',
     ...:                           '1/2/2000','1/3/2000'])

In [147]: dup_ts = pd.Series(np.arange(5), index = dates)

In [148]: dup_ts
Out[148]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int32
```

is_unique 속성을 통해 확인해보면 색인이 유일하지 않음을 알 수 있다.

```
In [150]: dup_ts.index.is_unique
Out[150]: False
```

이 시계열 데이터를 인덱싱하면 타임스탬프의 중복 여부에 따라 스칼라값이나 슬라이스가 생성된다.

```
In [151]: dup_ts['1/3/2000']
Out[151]: 4

In [152]: dup_ts['1/2/2000']
Out[152]:
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int32
```

유일하지 않은 타임스탬프를 가지는 데이터를 집계한다고 해보자. 한 가지 방법은 groupby에 level = 0 (단일 단계 인덱싱)을 넘기는 것이다.

```
In [155]: grouped.mean()
Out[155]:
2000-01-01    0
2000-01-02    2
2000-01-03    4
dtype: int32

In [157]: grouped.count()
Out[157]:
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

11.3 날자 범위, 빈도, 이동

pandas에서 일반적인 시계열은 불규칙적인 것으로 간주된다. 즉, 고정된 빈도를 갖지 않는다. 대부분의 애플리케이션에서 이는 충분하다. 하지만 시계열 안에서 누락된 값이 발생할지라도 일별, 월별 혹은 매 15분 같은 상대적인 고정 빈도에서의 작업이 요구되는 경우가 종종 있다. 다행스럽게도 pandas에는 리샘플링, 표준 시계열 빈도 모음, 빈도 추론 그리고 고정된 빈도의 날짜 범위를 위한 도구가 있다. 예를 들어 아래 예제 시계열을 고정된 일 빈도로 반환하려면 resample 메서드를 사용하면 된다.

```
In [112]: ts
Out[112]:
2011-01-02   -1.699012
2011-01-05   -0.215649
2011-01-07    2.473118
2011-01-08    0.259266
2011-01-10   -1.022399
2011-01-12    1.284342
dtype: float64

In [163]: resampler = ts.resample('D')

In [169]: for a,b in resampler:
     ...:     print(a,b)
     ...:
```

```
     ...:
2011-01-02 00:00:00 2011-01-02   -1.699012
dtype: float64
2011-01-03 00:00:00 Series([], dtype: float64)
2011-01-04 00:00:00 Series([], dtype: float64)
2011-01-05 00:00:00 2011-01-05   -0.215649
dtype: float64
2011-01-06 00:00:00 Series([], dtype: float64)
2011-01-07 00:00:00 2011-01-07    2.473118
dtype: float64
2011-01-08 00:00:00 2011-01-08    0.259266
dtype: float64
2011-01-09 00:00:00 Series([], dtype: float64)
2011-01-10 00:00:00 2011-01-10   -1.022399
dtype: float64
2011-01-11 00:00:00 Series([], dtype: float64)
2011-01-12 00:00:00 2011-01-12    1.284342
dtype: float64
```

문자열 'D' 는 일 빈도로 해석된다.

빈도 간 변환이나 **리샘플링**은 큰 주제이므로 다음에 따로 다룬다.

11.3.1 날짜 범위 생성하기

pandas.date_range를 사용하면 특정 빈도에 따라 지정한 길이만큼의 DatetimeIndex를 생성한다는 사실을 눈치 챘을 것이다.

```
In [170]: index = pd.date_range('2011-01-01', '2011-02-02')

In [171]: index
Out[171]:
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
               '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
               '2011-01-09', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-15', '2011-01-16',
               '2011-01-17', '2011-01-18', '2011-01-19', '2011-01-20',
               '2011-01-21', '2011-01-22', '2011-01-23', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28',
               '2011-01-29', '2011-01-30', '2011-01-31', '2011-02-01',
               '2011-02-02'],
              dtype='datetime64[ns]', freq='D')
```

기본적으로 date_range는 일별 타임스탬프를 생성한다. 만약 시작 날짜나 종료 날짜만 넘긴다면 생성할 기간의 숫자를 함께 전달해야 한다.

```
In [172]: pd.date_range('2010-01-03', periods = 30)
          #pd.date_range(start = '2010-01-03', periods = 30)
Out[172]:
DatetimeIndex(['2010-01-03', '2010-01-04', '2010-01-05', '2010-01-06',
               '2010-01-07', '2010-01-08', '2010-01-09', '2010-01-10',
               '2010-01-11', '2010-01-12', '2010-01-13', '2010-01-14',
               '2010-01-15', '2010-01-16', '2010-01-17', '2010-01-18',
               '2010-01-19', '2010-01-20', '2010-01-21', '2010-01-22',
               '2010-01-23', '2010-01-24', '2010-01-25', '2010-01-26',
               '2010-01-27', '2010-01-28', '2010-01-29', '2010-01-30',
               '2010-01-31', '2010-02-01'],
```

```
               dtype='datetime64[ns]', freq='D')

In [173]: pd.date_range(end ='2010-01-03', periods = 30)
Out[173]:
DatetimeIndex(['2009-12-05', '2009-12-06', '2009-12-07', '2009-12-08',
               '2009-12-09', '2009-12-10', '2009-12-11', '2009-12-12',
               '2009-12-13', '2009-12-14', '2009-12-15', '2009-12-16',
               '2009-12-17', '2009-12-18', '2009-12-19', '2009-12-20',
               '2009-12-21', '2009-12-22', '2009-12-23', '2009-12-24',
               '2009-12-25', '2009-12-26', '2009-12-27', '2009-12-28',
               '2009-12-29', '2009-12-30', '2009-12-31', '2010-01-01',
               '2010-01-02', '2010-01-03'],
              dtype='datetime64[ns]', freq='D')
```

시작과 종료 날짜는 생성된 날짜 색인에 대해 엄격한 경계를 정의한다. 예를 들어 날짜 색인이 각 월의 마지막 영업일을 포함하고 싶다면 빈도값으로 'BM' (월 영업마감일(Business Month End))을 전달할 것이다. 그러면 이 기간 안에 들어오는 날짜들만 포함된다.

```
In [174]: pd.date_range('2000-01-01', '2000-12-31', freq = 'BM')
Out[174]:
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', '2000-12-29'],
              dtype='datetime64[ns]', freq='BM')
```

date_range는 기본적으로 시작 시간이나 종료 시간의 타임스탬프를 보존한다.

```
In [176]: pd.date_range('2000-01-01 12:13:15', periods = 10)
Out[176]:
DatetimeIndex(['2000-01-01 12:13:15', '2000-01-02 12:13:15',
               '2000-01-03 12:13:15', '2000-01-04 12:13:15',
               '2000-01-05 12:13:15', '2000-01-06 12:13:15',
               '2000-01-07 12:13:15', '2000-01-08 12:13:15',
               '2000-01-09 12:13:15', '2000-01-10 12:13:15'],
              dtype='datetime64[ns]', freq='D')
```

가끔은 시간 정보를 포함하여 시작 날짜와 종료 날짜를 갖고 있으나 관례에 따라

```
In [6]: pd.date_range('2001-12-12 11:44:12', periods = 5, normalize = T
   ...: rue)
Out[6]:
DatetimeIndex(['2001-12-12', '2001-12-13', '2001-12-14', '2001-12-15',
               '2001-12-16'],
              dtype='datetime64[ns]', freq='D')
```

11.3.2 빈도와 날짜 오프셋

padas에서 빈도는 **기본 빈도(base frequency)** 와 배수의 조합으로 이뤄진다. 기본 빈도는 보통 'M'(월별), 'H'(시간별) 처럼 짧은 문자열로 참조된다. 각 기본 빈도에는 일반적으로 **날짜 오프셋**이라고 불리는 객체를 사용할 수 있다. 예를 들어 시간별 빈도는 Hour 클래스를 사용해서 표현할 수 있다.

```
In [7]: from pandas.tseries.offsets import Hour, Minute

In [8]: hour = Hour()

In [9]: hour
Out[9]: <Hour>
```

이 오프셋의 곱은 정수를 넘겨서 구할 수 있다.

```
In [10]: four_hour = Hour(4)

In [11]: four_hour
Out[11]: <4 * Hours>
```

대부분의 애플리케이션에서는 이런 객체들을 직접 만들어야 할 경우는 절대 없겠지만 대신 'H' 또는 '4H' 처럼 문자열로 표현하게 될 것이다.

```
In [13]: pd.date_range('2001-01-01', '2001-01-02 12:00:00', freq = '4h'
    ...: )
Out[13]:
DatetimeIndex(['2001-01-01 00:00:00', '2001-01-01 04:00:00',
               '2001-01-01 08:00:00', '2001-01-01 12:00:00',
               '2001-01-01 16:00:00', '2001-01-01 20:00:00',
               '2001-01-02 00:00:00', '2001-01-02 04:00:00',
               '2001-01-02 08:00:00', '2001-01-02 12:00:00'],
              dtype='datetime64[ns]', freq='4H')

In [14]: pd.date_range('2001-01-01', '2001-01-02 12:00:00', freq = '1h3
    ...: 0min')
Out[14]:
DatetimeIndex(['2001-01-01 00:00:00', '2001-01-01 01:30:00',
               '2001-01-01 03:00:00', '2001-01-01 04:30:00',
               '2001-01-01 06:00:00', '2001-01-01 07:30:00',
               '2001-01-01 09:00:00', '2001-01-01 10:30:00',
               '2001-01-01 12:00:00', '2001-01-01 13:30:00',
               '2001-01-01 15:00:00', '2001-01-01 16:30:00',
               '2001-01-01 18:00:00', '2001-01-01 19:30:00',
               '2001-01-01 21:00:00', '2001-01-01 22:30:00',
               '2001-01-02 00:00:00', '2001-01-02 01:30:00',
               '2001-01-02 03:00:00', '2001-01-02 04:30:00',
               '2001-01-02 06:00:00', '2001-01-02 07:30:00',
               '2001-01-02 09:00:00', '2001-01-02 10:30:00',
               '2001-01-02 12:00:00'],
              dtype='datetime64[ns]', freq='90T')

In [15]: pd.date_range('2001-01-02', periods = 10, freq = '2h30min')
Out[15]:
DatetimeIndex(['2001-01-02 00:00:00', '2001-01-02 02:30:00',
               '2001-01-02 05:00:00', '2001-01-02 07:30:00',
               '2001-01-02 10:00:00', '2001-01-02 12:30:00',
               '2001-01-02 15:00:00', '2001-01-02 17:30:00',
               '2001-01-02 20:00:00', '2001-01-02 22:30:00'],
              dtype='datetime64[ns]', freq='150T')
```

어떤 빈도는 시간상에서 균일하게 자리 잡고 있지 않은 경우도 있다. 예를 들어 'M' (월 마지막 일)은 월중 일수에 의존적이며 'BM(월 영업마감일)' 은 월말은 주말인지 아닌지에 따라 다르다.

이를 표현할 수 있는 적당한 용어가 없어서 나는 이를 **앵커드** 오프셋이라고 부른다.

월별 주차

한 가지 유용한 빈도 클래스는 WOM으로 시작하는 '월별 주차'다. 월별 주차를 사용하면 매월 3째주 금요일 같은 날짜를 얻을 수 있다.

```
In [16]: rng = pd.date_range('2012-01-07', '2013-01-05', freq = 'WOM-3F
    ...: RI')

In [17]: rng
Out[17]:
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',
               '2012-09-21', '2012-10-19', '2012-11-16', '2012-12-21'],
              dtype='datetime64[ns]', freq='WOM-3FRI')
```

11.3.3 데이터 시프트

시프트는 데이터를 시간 축에서 앞이나 뒤로 이동하는 것을 의미한다. Series와 DateFrame은 색인을 변경하지 않고 데이터를 앞이나 뒤로 느슨한 시프트를 수행하는 shift 메서드를 가지고 있다.

```
In [21]: ts = pd.Series(np.random.randn(4),
    ...:                    index = pd.date_range('2011-01-02',
    ...:                    periods =4,
    ...:                    freq = 'M' ))

In [22]: ts
Out[22]:
2011-01-31   -0.542971
2011-02-28    0.242877
2011-03-31    0.477533
2011-04-30    0.429941
Freq: M, dtype: float64

In [23]: ts.shift(2)
Out[23]:
2011-01-31         NaN
2011-02-28         NaN
2011-03-31   -0.542971
2011-04-30    0.242877
Freq: M, dtype: float64

In [24]: ts.shift(-2)
Out[24]:
2011-01-31    0.477533
2011-02-28    0.429941
2011-03-31         NaN
2011-04-30         NaN
Freq: M, dtype: float64
```

이렇게 시프트를 하면 시계열의 시작이나 끝이 결측치로 나온다.

shift는 일반저긍로 한 시계열 내에서, 혹은 DataFrame의 컬럼으로 표현할 수 있는 여러 시계열에서의 퍼센트 변화를 계산할 때 흔히 사용하며, 코드로는 다음과 같이 표현한다.

```
ts/ts.shift(1) -1
```

느슨한 시프트는 색인을 바꾸지 않기에 어떤 데이터는 버려지기도 한다. 그래서 만약 빈도를 알고 있다면 shift에 빈도를 넘겨서 타임스탬프가 확장되도록 할 수 있다.

```
In [25]: ts.shift(2, freq = 'M')
Out[25]:
2011-03-31   -0.542971
2011-04-30    0.242877
2011-05-31    0.477533
2011-06-30    0.429941
Freq: M, dtype: float64

In [26]: ts.shift(3, freq = 'D')
Out[26]:
2011-02-03   -0.542971
2011-03-03    0.242877
2011-04-03    0.477533
2011-05-03    0.429941
dtype: float64

In [27]: ts.shift(1, freq = '90T')
Out[27]:
2011-01-31 01:30:00   -0.542971
2011-02-28 01:30:00    0.242877
2011-03-31 01:30:00    0.477533
2011-04-30 01:30:00    0.429941
Freq: M, dtype: float64
#T는 분을 의미한다.
```

오프셋만큼 날짜 시프트하기

pandas의 날짜 오프셋은 datetime 이나 Timestamp 객체에서도 사용할 수 있다.

```
In [28]: from pandas.tseries.offsets import Day, MonthEnd

In [29]: now = datetime(2001,1,5)

In [30]: now + 3 * Day()
Out[30]: Timestamp('2001-01-08 00:00:00')
```

만일 MonthEnd 같은 앵커드 오프셋을 추가한다면 빈도 규칙의 다음 날짜로 롤 포워드 된다.

```
In [30]: now + 3 * Day()
Out[30]: Timestamp('2001-01-08 00:00:00')

In [31]: now + MonthEnd()
Out[31]: Timestamp('2001-01-31 00:00:00')

In [32]: now + MonthEnd(2)
Out[32]: Timestamp('2001-02-28 00:00:00')
```

앵커드 오프셋은 rollforward와 rollback 메서드를 사용해서 명시적으로 각각 날자를 앞으로 밀거나 뒤로 당길 수 있다.

```
In [33]: offset = MonthEnd()

In [34]: offset.rollforward(now)
Out[34]: Timestamp('2001-01-31 00:00:00')

In [36]: offset.rollback(now)
Out[36]: Timestamp('2000-12-31 00:00:00')
```

이 메서드를 groupby와 함께 사용하면 날짜 오프셋을 영리하게 사용할 수 있다.

```
In [39]: ts = pd.Series(np.random.randn(10),
                index = pd.date_range('2000-01-01',
    ...:                                      periods = 10,
    ...:                                      freq = '4d')
    ...:                         )

In [40]: ts
Out[40]:
2000-01-01   -0.275219
2000-01-05   -1.735932
2000-01-09   -1.735604
2000-01-13    0.564264
2000-01-17    0.517631
2000-01-21   -0.067098
2000-01-25   -0.349571
2000-01-29   -1.784134
2000-02-02    1.335372
2000-02-06    0.854501
Freq: 4D, dtype: float64

In [41]: ts.groupby(offset.rollforward).mean()
Out[41]:
2000-01-31   -0.608208
2000-02-29    1.094937
dtype: float64
```

물론 가장 쉽고 빠른 방법은 resample을 사용하는 것이다.

```
In [43]: ts.resample('M').mean()
Out[43]:
2000-01-31   -0.608208
2000-02-29    1.094937
Freq: M, dtype: float64
```

11.4 시간대 다루기

시간대를 처리하는 일은 시계열을 다루는 작업 중에서 가장 유쾌하지 않은 부분 중 하나다. 특히 일광절약시간(DST, 섬머타임)은 문제를 일으키는 흔한 요인 중 하나다. 시계열을 다루는 많은 사용자는 현재 국제 표준이며 그리니치 표준시를 계승하는 **국제표준시**를 선택한다. 시간대는 UTC로부터 떨어진 오프셋으로 표현되는데 예를 들면 뉴욕은 일광절약시간일 때 UTC 보다 4시간이 늦으며 아닐 때는 5시간이 늦는다.

파이썬에서 시간대 정보는 전 세계의 시간대 정보를 모아둔 **올슨 데이터베이스**를 담고 있는 서드파티 라이브러리인 pytz에서 얻어온다. 이는 특히 역사적인 데이터를 다룰 때 중요한데 DST 날짜 (그리고 심지어는 UTC 오프셋마저)는 지역 정부의 변덕에 따라 여러 차례 변경되었기 때문이다.

시간대 이름은 문서와 파이썬 셸에서 직접 확인할 수 있다.

```
In [49]: pytz.common_timezones[-5:]
Out[49]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

pytz에서 시간대 객체를 얻으려면 pytz.timezone을 사용하면 된다.

11.4.1 시간대 지역화와 변환

기본적으로 pandas에서 시계열은 **시간대를 엄격히 다루지 않는다.**

```
In [54]: rng = pd.date_range('2011-12-11', periods = 6, freq='D')

In [56]: ts = pd.Series(np.random.randn(len(rng)),
    ...:                 index = rng)

In [57]: ts
Out[57]:
2011-12-11    0.545912
2011-12-12   -1.002333
2011-12-13   -0.067111
2011-12-14   -0.096603
2011-12-15    0.685956
2011-12-16   -0.261268
Freq: D, dtype: float64
```

색인의 tz필드는 None이다.

```
In [58]: print(ts.index.tz)
None
```

시간대를 지정해서 날짜 범위를 생성할 수 있다.

```
In [61]: pd.date_range('2011-12-31', periods = 10, freq = 'D', tz = 'As
    ...: ia/Seoul')
Out[61]:
DatetimeIndex(['2011-12-31 00:00:00+09:00', '2012-01-01 00:00:00+09:00',
               '2012-01-02 00:00:00+09:00', '2012-01-03 00:00:00+09:00',
               '2012-01-04 00:00:00+09:00', '2012-01-05 00:00:00+09:00',
               '2012-01-06 00:00:00+09:00', '2012-01-07 00:00:00+09:00',
               '2012-01-08 00:00:00+09:00', '2012-01-09 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Seoul]', freq='D')
```

지역화시간으로의 변환은 tz_localize 메서드로 처리할 수 있다.

```
In [62]: ts
Out[62]:
2011-12-11    0.545912
2011-12-12   -1.002333
2011-12-13   -0.067111
2011-12-14   -0.096603
```

```
2011-12-15    0.685956
2011-12-16   -0.261268
Freq: D, dtype: float64

In [63]: ts_utc = ts.tz_localize('UTC')

In [64]: ts_utc
Out[64]:
2011-12-11 00:00:00+00:00    0.545912
2011-12-12 00:00:00+00:00   -1.002333
2011-12-13 00:00:00+00:00   -0.067111
2011-12-14 00:00:00+00:00   -0.096603
2011-12-15 00:00:00+00:00    0.685956
2011-12-16 00:00:00+00:00   -0.261268
Freq: D, dtype: float64

In [79]: ts_HONGKONG = ts_utc.tz_convert('Asia/Hong_Kong')

In [80]: ts_HONGKONG
Out[80]:
2011-12-11 08:00:00+08:00    0.545912
2011-12-12 08:00:00+08:00   -1.002333
2011-12-13 08:00:00+08:00   -0.067111
2011-12-14 08:00:00+08:00   -0.096603
2011-12-15 08:00:00+08:00    0.685956
2011-12-16 08:00:00+08:00   -0.261268
Freq: D, dtype: float64

In [81]: ts_HONGKONG.index
Out[81]:
DatetimeIndex(['2011-12-11 08:00:00+08:00', '2011-12-12 08:00:00+08:00',
               '2011-12-13 08:00:00+08:00', '2011-12-14 08:00:00+08:00',
               '2011-12-15 08:00:00+08:00', '2011-12-16 08:00:00+08:00'],
              dtype='datetime64[ns, Asia/Hong_Kong]', freq='D')
```

시계열이 특정 시간대로 지역화되고 나면 tz_convert를 이용해서 다른 시간대로 변환 가능하다.

```
In [83]: ts_utc.tz_convert('America/New_York')
Out[83]:
2011-12-10 19:00:00-05:00    0.545912
2011-12-11 19:00:00-05:00   -1.002333
2011-12-12 19:00:00-05:00   -0.067111
2011-12-13 19:00:00-05:00   -0.096603
2011-12-14 19:00:00-05:00    0.685956
2011-12-15 19:00:00-05:00   -0.261268
Freq: D, dtype: float64
```

위 시계열의 경우에는 America/New_York 시간대에서 일광절약시간을 사용하고 있는데, 동부 표준시(EST)로 맞춘 다음 UTC 혹은 베를린 시간으로 변환할 수 있다.

11.4.2 시간대를 고려해서 Timestamp 객체 다루기

시계열이나 날짜 범위와 비슷하게 개별 Timestamp 객체도 시간대를 고려한 형태로 변환이 가능하다.

```
In [84]: stamp = pd.Timestamp('2001-11-12 12:13:41')

In [85]: stamp
Out[85]: Timestamp('2001-11-12 12:13:41')

In [86]: stamp_utc = stamp.tz_localize('utc')

In [87]: stamp_utc.tz_convert('Asia/Seoul')
Out[87]: Timestamp('2001-11-12 21:13:41+0900', tz='Asia/Seoul')
```

Timestamp 객체를 생성할 때 시간대를 직접 넘겨주는 것도 가능하다.

```
In [88]: stamp_moscow = pd.Timestamp('2001-01-24 12:53:23',
    ...:                             tz='Europe/Moscow')

In [89]: stamp_moscow
Out[89]: Timestamp('2001-01-24 12:53:23+0300', tz='Europe/Moscow')
```

시간대를 고려한 Timestamp 객체는 내부적으로 UTC 타임스탬프 값을 유닉스 에포크 (1970년 1월 1일)부터 현재까지 나노초로 저장하고 있다. 이 UTC 값은 시간대 변환 과정에서 변하지 않고 유지된다.

```
In [95]: stamp_utc = pd.Timestamp('1970-01-02')

In [96]: stamp_utc.value
Out[96]: 86400000000000

In [98]: stamp_utc.tz_localize('America/New_York')
Out[98]: Timestamp('1970-01-02 00:00:00-0500', tz='America/New_York')

In [99]: stamp_NY = stamp_utc.tz_localize('America/New_York')

In [100]: stamp_NY.value
Out[100]: 104400000000000
```

pandas의 DateOffset 객체를 이용해서 시간 연산을 수행할 때는 가능하다면 일광절약시간을 고려한다. DST로 전환되기 직전의 타임스탬프에 대한 예제를 살펴보자. 먼저 DST 시행 30분전의 Timestamp를 생성하자.

```
In [101]: from pandas.tseries.offsets import Hour

In [102]: stamp = pd.Timestamp('2000-12-12 01:30', tz ='Asia/Seoul' )

In [103]: stamp
Out[103]: Timestamp('2000-12-12 01:30:00+0900', tz='Asia/Seoul')

In [104]: stamp + Hour()
Out[104]: Timestamp('2000-12-12 02:30:00+0900', tz='Asia/Seoul')
```

그리고 DST 시행 90분 전의 Timestamp를 생성하자.

11.4.2 다른 시간대 간의 연산

서로 다른 시간대를 갖는 두 시계열이 하나로 합쳐지면 결과는 UTC가 된다. 타임스탬프는 내부적으료 UTC로 저장되므로 추가적인 변환이 불필요한 명료한 연산이다.

```
In [105]: rng = pd.date_range('2011-01-14', periods = 10, freq = 'B')

In [106]: ts = pd.Series(np.random.randn(len(rng)), index = rng)

In [107]: ts
Out[107]:
2011-01-14    0.454879
2011-01-17    0.908793
2011-01-18    0.732212
2011-01-19    1.384150
2011-01-20   -1.183462
2011-01-21   -0.955555
2011-01-24   -1.730386
2011-01-25   -1.156773
2011-01-26    0.008288
2011-01-27   -0.613690
Freq: B, dtype: float64

In [110]: ts1 = ts[:7].tz_localize('Europe/London')

In [111]: ts2 = ts1[:7].tz_convert('Europe/Moscow')

In [113]: result = ts1 + ts2

In [114]: result.index
Out[114]:
DatetimeIndex(['2011-01-14 00:00:00+00:00', '2011-01-17 00:00:00+00:00',
               '2011-01-18 00:00:00+00:00', '2011-01-19 00:00:00+00:00',
               '2011-01-20 00:00:00+00:00', '2011-01-21 00:00:00+00:00',
               '2011-01-24 00:00:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

표 11-4 기본 시계열 빈도

축약	오프셋 종류	설명
D	Day	달력상의 일
B	BusinessDay	매 영업일
H	Hour	매시
T 또는 min	Minute	매분
S	Second	매초
L 또는 ms	Milli	밀리초(1/1000초)
U	Micro	마이크로초(1/1,000,000초)
M	MonthEnd	월 마지막 일
BM	BusinessMonthEnd	월 영업마감일
MS	MonthBegin	월 시작일
BMS	BusinessMonthBegin	월 영업시작일
W-MON, W-TUE, ...	Week	요일. MON, TUE, WED, THU, FRI, SAT, SUN
WOM-1MON, WOM-2MON, ...	WeekOfMonth	월별 주차와 요일. 예를 들어 WOM-3FRI는 매월 3째 주 금요일이다.
Q-JAN, Q-FEB, ...	QuarterEnd	지정된 월을 해당년도의 마감으로 하며 지정된 월의 마지막 날짜를 가리키는 분기 주기(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	지정된 월을 해당년도의 마감으로 하며 지정된 월의 마지막 영업일을 가리키는 분기 주기
QS-JAN, QS-FEB, ...	QuarterBegin	지정된 월을 해당년도의 마감으로 하며 지정된 월의 첫째 날을 가리키는 분기 주기
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	지정된 월을 해당년도의 마감으로 하며 지정된 월의 첫 번째 영업일을 가리키는 분기 주기
A-JAN, A-FEB, ...	YearEnd	주어진 월의 마지막 일을 가리키는 연간 주기(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)

11.5 기간과 기간 연산

며칠, 몇 개월, 몇 분기, 몇 해 같은 기간은 Period 클래스로 표현할 수 있으며 문자열이나 정수 그리고 빈도 옵션으로 생성한다.

```
In [115]: p = pd.Period(2007 , freq = 'A-DEC')

In [116]: p
Out[116]: Period('2007', 'A-DEC')
```

여기서 Periods 객체는 2007년 1월 1일 부터 같은 해 12월 31일 까지의 기간을 표현한다. 이 기간에 정수를 더하거나 빼서 편리하게 정해진 빈도에 따라 기간을 이동시킬 수 있다.

```
In [117]: p +5
Out[117]: Period('2012', 'A-DEC')

In [118]: p - 2
Out[118]: Period('2005', 'A-DEC')
```

만약 두 기간이 같은 빈도를 가진다면 두 기간의 차는 둘 사이의 간격이 된다.

```
In [119]: pd.Period('2014', 'A-DEC') - p
Out[119]: <7 * YearEnds: month=12>
```

일반적인 기간 범위는 period_range 함수로 생성할 수 있다.

```
In [121]: rng = pd.period_range('2011-11-11','2013-11-15', freq = 'M')

In [122]: rng
Out[122]:
PeriodIndex(['2011-11', '2011-12', '2012-01', '2012-02', '2012-03', '2012-04',
             '2012-05', '2012-06', '2012-07', '2012-08', '2012-09', '2012-10',
             '2012-11', '2012-12', '2013-01', '2013-02', '2013-03', '2013-04',
             '2013-05', '2013-06', '2013-07', '2013-08', '2013-09', '2013-10',
             '2013-11'],
            dtype='period[M]', freq='M')
```

PeriodIndex 클래스는 순차적인 기간을 저장하며 다른 pandas 자료구조에서 축 색인과 마찬가지로 사용된다.

```
In [124]: pd.Series(np.random.randn(len(rng)), index = rng)
Out[124]:
2011-11   -0.619060
2011-12   -0.289394
2012-01   -0.363224
2012-02    0.748041
2012-03    0.338265
2012-04   -0.082514
2012-05    1.214453
2012-06   -0.673887
2012-07    0.203180
2012-08    1.035568
2012-09    0.437126
2012-10   -0.446544
2012-11    0.996515
2012-12    1.482678
2013-01   -0.534774
2013-02    0.333657
2013-03   -0.786500
2013-04    0.885762
2013-05   -0.958771
2013-06    0.497379
2013-07   -0.718429
2013-08    0.237656
2013-09   -0.075318
2013-10    0.238408
2013-11    0.405044
Freq: M, dtype: float64
```

다음과 같은 문자열 배열을 이용해서 PeriodIndex 클래스를 생성하는 것도 가능하다.

```
In [125]: values = ['2001Q1', '2001Q4', '2012Q1']

In [126]: index = pd.PeriodIndex(values, freq = 'Q-DEC')

In [127]: index
Out[127]: PeriodIndex(['2001Q1', '2001Q4', '2012Q1'],
dtype='period[Q-DEC]', freq='Q-DEC')
```

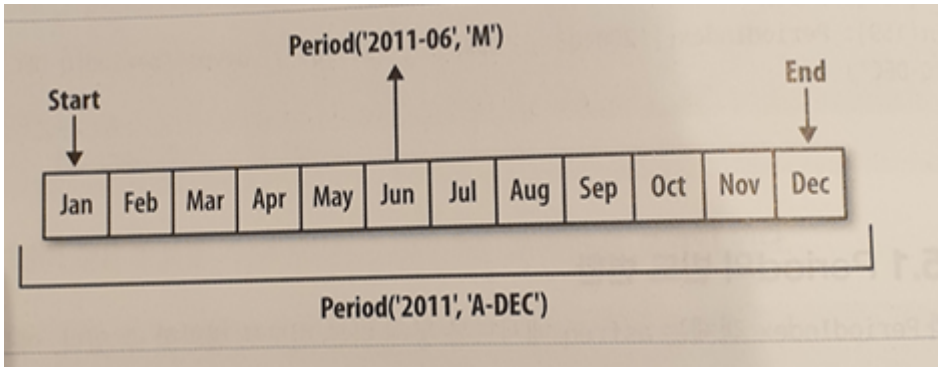
11.5.1 Periods의 빈도 변환

기관과 PeriodIndex 객체는 asfreq 메서드를 통해 다른 빈도로 변환할 수 있다. 예를 들어 새해 첫날부터 시작하는 연간 빈도를 월간 빈도로 변환해보자. 꽤간단하다.

```
In [128]: p = pd.Period('2007', freq = 'A-DEC')

In [129]: p
Out[129]: Period('2007', 'A-DEC')

In [130]: p.asfreq('M', how = 'start')
Out[130]: Period('2007-01', 'M')

In [131]: p.asfreq('M', how = 'end')
Out[131]: Period('2007-12', 'M')
```

Period('2007', 'A-DEC')는 전체 기간에 대한 커서로 생각할 수 있고 월간으로 다시 나눌 수 있다. 이 내용은 다음 그림을 참조하자. **회계연도** 마감이 12월이 아닌 경우에는 월간 빈도가 달라지낟.

```
In [136]: p = pd.Period('2007', freq = 'A-JUN')

In [137]: p
Out[137]: Period('2007', 'A-JUN')

In [138]: p.asfreq('M', how = 'start')
Out[138]: Period('2006-07', 'M')

In [139]: p.asfreq('M', how = 'end')
Out[139]: Period('2007-06', 'M')
```



Period의 빈도 변환

빈도가 상위 단계에서 하위 단계로 변환되는 경우 상위 기간은 하위 기간이 어디에 속했는지에 따라 결정된다. 예를 들어 A-JUN 빈도 일 경우 2007년 8월은 실제로 2008년 기간에 속한다.

```
In [140]: p = pd.Period('Aug-2007', 'M')

In [142]: p.asfreq('A-JUN')
Out[142]: Period('2008', 'A-JUN')
```

모든 PeriodIndex 객체나 시계열은 지금까지 살펴본 내용과 같은 방식으로 변환할 수 있다.

```
In [143]: rng = pd.period_range('2001', '2004', freq = 'A-DEC')

In [144]: rng1 = pd.date_range('2001', '2004', freq = 'A-DEC')

In [145]: rng1
Out[145]: DatetimeIndex(['2001-12-31', '2002-12-31', '2003-12-31'],
                        dtype='datetime64[ns]', freq='A-DEC')

In [146]: rng
Out[146]: PeriodIndex(['2001', '2002', '2003', '2004'], dtype='period[A-DEC]'
                      , freq='A-DEC')

In [147]: ts = pd.Series(np.random.rand(len(rng)), index = rng)

In [148]: ts
Out[148]:
2001    0.876758
2002    0.537443
2003    0.336231
2004    0.111649
Freq: A-DEC, dtype: float64


In [149]: ts.asfreq('M', how = 'start')
Out[149]:
2001-01    0.876758
2002-01    0.537443
2003-01    0.336231
2004-01    0.111649
Freq: M, dtype: float64

In [150]: ts.asfreq('M', how = 'end')
Out[150]:
2001-12    0.876758
2002-12    0.537443
2003-12    0.336231
2004-12    0.111649
Freq: M, dtype: float64
```

위 예제에서 연 빈도는 해당 빈도의 시작 월 부터 시작하는 월 빈도로 치환된다. 만일 매해의 마지막 영업일을 대신 사용하고 싶다면 'B' 빈도를 사용하고 해당 기간의 종료 지점을 지정해서 변환할 수 있다.

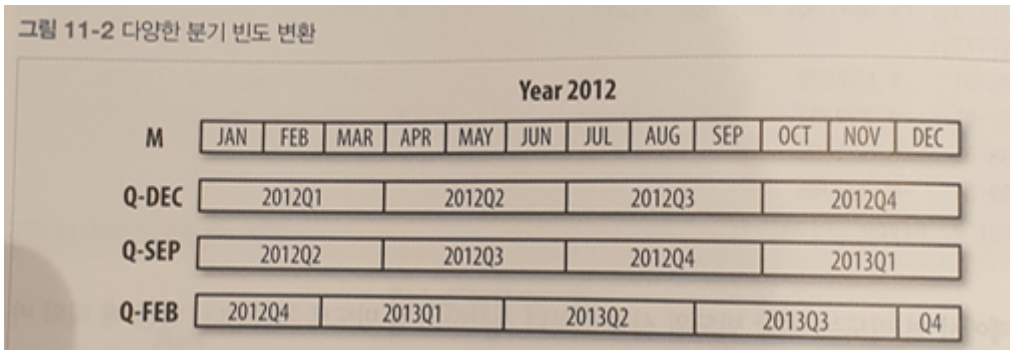
```
In [151]: ts.asfreq('B', how = 'end')
Out[151]:
2001-12-31    0.876758
2002-12-31    0.537443
2003-12-31    0.336231
2004-12-31    0.111649
Freq: B, dtype: float64
```

11.5.2 분기 빈도

분기 데이터는 재정, 금융 및 다른 분야에서 표준으로 사용된다. 많은 분기 데이터는 일반저긍로 **회계연도의 끝**인 12월의 마지막 날이나 마지막 업무일을 기준으로 보고하는데, 2012Q4는 회계연도의 끝이 어딘가에 따라 의미가 달라진다. pandas는 12가지 모든 경우의 수를 지원하며 분기 빈도는 Q-JAN 부터 Q-DEC 까지다.

```
In [155]: p = pd.Period('2012Q4', freq = 'Q-JAN')

In [156]: p
Out[156]: Period('2012Q4', 'Q-JAN')
```

회계연도 마감이 1월인 경우라면 2012Q4는 11월부터 1월 까지가 되고 일간 빈도로 검사할 수 있다.




```
In [157]: p.asfreq('D','start')
Out[157]: Period('2011-11-01', 'D')

In [158]: p.asfreq('D','end')
Out[158]: Period('2012-01-31', 'D')
```

이렇게 하여 기간 연산을 매우 쉽게 할 수 있는데, 그 예로 분기 영업 마감일의 오후 4시를 가리키는 타임스탬프는 다음과 같이 구할 수 있다.

```
In [160]: p4pm = (p.asfreq('B','e') -1).asfreq('T','s')+16*60

In [161]: p4pm
Out[161]: Period('2012-01-30 16:00', 'T')

In [162]: p4pm.to_timestamp()
Out[162]: Timestamp('2012-01-30 16:00:00')
```

period_range를 사용해서 분기 범위를 생성할 수 있다. 연산 역시 동일한 방법으로 수행할 수 있다.

```
In [166]: rng = pd.period_range('2011Q3','2012Q4',freq='Q-JAN')

In [167]: ts=pd.Series(len(rng), index = rng)
```

```
In [169]: ts=pd.Series(np.random.rand(len(rng)), index = rng)

In [170]: ts
Out[170]:
2011Q3    0.907126
2011Q4    0.521928
2012Q1    0.742491
2012Q2    0.827241
2012Q3    0.158446
2012Q4    0.851927
Freq: Q-JAN, dtype: float64

In [171]: new_rng = (rng.asfreq('B','e')-1).asfreq('T','s')+16*60

In [172]: new_rng
Out[172]:
PeriodIndex(['2010-10-28 16:00', '2011-01-28 16:00', '2011-04-28 16:00',
             '2011-07-28 16:00', '2011-10-28 16:00', '2012-01-30 16:00'],
            dtype='period[T]', freq='T')

In [173]: ts.index = new_rng.to_timestamp()

In [174]: new_rng
Out[174]:
PeriodIndex(['2010-10-28 16:00', '2011-01-28 16:00', '2011-04-28 16:00',
             '2011-07-28 16:00', '2011-10-28 16:00', '2012-01-30 16:00'],
            dtype='period[T]', freq='T')

In [175]: ts
Out[175]:
2010-10-28 16:00:00    0.907126
2011-01-28 16:00:00    0.521928
2011-04-28 16:00:00    0.742491
2011-07-28 16:00:00    0.827241
2011-10-28 16:00:00    0.158446
2012-01-30 16:00:00    0.851927
dtype: float64
```

11.5.3 타임스탬프와 기간 서로 변환하기

타임스탬프로 색인된 Series와 DataFrame 객체는 to_period 매서드를 사용해서 기간으로 변환가능하다.

```
In [177]: rng = pd.date_range('2000-01-25',periods = 3, freq='M')

In [178]: rng
Out[178]: DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31'],
        dtype='datetime64[ns]', freq='M')

In [179]: len(rng)
Out[179]: 3

In [180]: ts = pd.Series(np.random.rand(len(rng)),index = rng)

In [181]: ts
Out[181]:
2000-01-31    0.047780
```

```
2000-02-29    0.750813
2000-03-31    0.524179
Freq: M, dtype: float64

In [182]: pts = ts.to_period()

In [183]: pts
Out[183]:
2000-01    0.047780
2000-02    0.750813
2000-03    0.524179
Freq: M, dtype: float64
```

여기서 말하는 기간은 겹치지 않는 시간상의 간격을 뜻하므로 주어진 빈도에서 타임스탬프는 하나의 기간에만 속한다. 새로운 PeriodIndex의 빈도는 기본적으로 타임스탬프 값을 통해 추론되지만 원하는 빈도를 직접 지정할 수도 있다. 결과에 중복되는 기간이 나오더라도 문제가 되지 않는다.

```
In [184]: rng = pd.date_range('1/29/2000', periods = 6, freq='D')

In [185]: ts2 = pd.Series(np.random.randn(6),index = rng)

In [186]: ts2
Out[186]:
2000-01-29    1.361311
2000-01-30    1.968949
2000-01-31   -0.768482
2000-02-01    1.202130
2000-02-02   -1.161523
2000-02-03    0.253703
Freq: D, dtype: float64

In [187]: ts2.to_period('M')
Out[187]:
2000-01    1.361311
2000-01    1.968949
2000-01   -0.768482
2000-02    1.202130
2000-02   -1.161523
2000-02    0.253703
Freq: M, dtype: float64

In [188]: ts2.to_period('Y')
Out[188]:
2000    1.361311
2000    1.968949
2000   -0.768482
2000    1.202130
2000   -1.161523
2000    0.253703
Freq: A-DEC, dtype: float64
```

기간을 타임스탬프로 변환하려면 tp_timestamp 메서드를 이용하면 된다.

```
In [191]: pts = ts2.to_period()

In [192]: pts
```

```
Out[192]:
2000-01-29    1.361311
2000-01-30    1.968949
2000-01-31   -0.768482
2000-02-01    1.202130
2000-02-02   -1.161523
2000-02-03    0.253703
Freq: D, dtype: float64

In [193]: pts.to_timestamp(how='e')
Out[193]:
2000-01-29 23:59:59.999999999    1.361311
2000-01-30 23:59:59.999999999    1.968949
2000-01-31 23:59:59.999999999   -0.768482
2000-02-01 23:59:59.999999999    1.202130
2000-02-02 23:59:59.999999999   -1.161523
2000-02-03 23:59:59.999999999    0.253703
Freq: D, dtype: float64
```

11.5.4 배열로 PeriodIndex 생성하기

고정된 빈도를 갖는 데이터는 종종 여러 컬럼에 걸쳐 기간에 대한 정보를 함께 저장하기도 한다. 예를 들어 거시경제학 데이터셋에는 연도와 분기가 구분된 컬럼에 존재한다.

```
In [196]: data = pd.read_csv('macrodata.csv')

In [197]: data.head()
Out[197]:
     year  quarter   realgdp  realcons  ...  unemp      pop  infl  realint
0  1959.0      1.0  2710.349    1707.4  ...    5.8  177.146  0.00     0.00
1  1959.0      2.0  2778.801    1733.7  ...    5.1  177.830  2.34     0.74
2  1959.0      3.0  2775.488    1751.8  ...    5.3  178.657  2.74     1.09
3  1959.0      4.0  2785.204    1753.7  ...    5.6  179.386  0.27     4.06
4  1960.0      1.0  2847.699    1770.5  ...    5.2  180.007  2.31     1.19

In [198]: data.year
Out[198]:
0      1959.0
1      1959.0
2      1959.0
3      1959.0
4      1960.0
        ...
198    2008.0
199    2008.0
200    2009.0
201    2009.0
202    2009.0
Name: year, Length: 203, dtype: float64

In [199]: data.quarter
Out[199]:
0      1.0
1      2.0
2      3.0
3      4.0
4      1.0
```

```
      ...
198    3.0
199    4.0
200    1.0
201    2.0
202    3.0
Name: quarter, Length: 203, dtype: float64
```

이 배열을 PeriodIndex에 빈도값과 전달하면 이를 조합해서 DataFrame에서 사용할 수 있는 색인을 만들어 낸다.

```
In [204]: index = pd.PeriodIndex(year = data.year,
     ...:                         quarter = data.quarter,
     ...:                         freq = 'Q-DEC')

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')

In [206]: data.index = index

In [207]: data.infl
Out[207]:
1959Q1    0.00
1959Q2    2.34
1959Q3    2.74
1959Q4    0.27
1960Q1    2.31
          ...
2008Q3   -3.16
2008Q4   -8.79
2009Q1    0.94
2009Q2    3.37
2009Q3    3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6 리샘플링과 빈도 변환

리샘플링은 시계열의 빈도를 변환하는 과정을 일컫는다. 상위 빈도의 데이터를 하위 빈도로 절계하는 것을 **다운샘플링**이라고 하며 반대 과정을 **업샘플링**이라고 한다. 모든 리샘플링이 이 두 가지 범주에 들어가지는 않는다. 예를 들어 W-WED(수요일을 기준으로 한 주간)를 W-FRI로 변경하는 것은 업 샘플링도, 다운샘플링도 아니다 .

pandas 객체는 resanmple 메서드를 가지고 있는데, 빈도 변환과 관련된 모든 작업에서 유용하게 사용되는 메서드다. resample은 groupby와 비슷한 API를 가지고 있는데 resample을 호출해서 데이터를 그룹짓고 요약함수를 적용하는 식이다.

```
In [208]: rng = pd.date_range('2000-12-13',periods = 100, freq = 'D')

In [209]: ts = pd.Series(np.random.rand(len(rng)),index = rng)
```

```
In [210]: ts
Out[210]:
2000-12-13    0.838668
2000-12-14    0.705476
2000-12-15    0.910889
2000-12-16    0.125732
2000-12-17    0.234818
                ...
2001-03-18    0.018822
2001-03-19    0.619678
2001-03-20    0.726637
2001-03-21    0.927905
2001-03-22    0.916734
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean()
Out[211]:
2000-12-31    0.496813
2001-01-31    0.503559
2001-02-28    0.437937
2001-03-31    0.558389
Freq: M, dtype: float64

In [212]: ts.resample('M',kind = 'period').mean()
Out[212]:
2000-12    0.496813
2001-01    0.503559
2001-02    0.437937
2001-03    0.558389
Freq: M, dtype: float64
```

resample은 유연한 고수준의 메서드로, 매우 큰 시계열 데이터를 처리할 수 있다.

인자	설명
freq	원하는 리샘플링 빈도를 가리키는 문자열이나 DateOffset('M','5min',Second(15))
axis	리샘플링을 수행할 축, 기본값은 axis = 0이다.
fill_method	업샘플링 시 사용할 보간 방법, 'ffilll', 'bfill' 이 있다. 기본값은 None이다.
closed	다운샘플링 시 각 간격의 어느 쪽으로 포함할지 가리킨다. 'right', 'left'가 있고, 기본값은 'right'이다.
label	다운샘플링 시 집계된 결과의 라벨을 결정한다. 'rigth'와 'lefr'가 있다.

11.6.1 다운샘플링

시계열 데이터를 규칙적인 하위 빈도로 집계하는 일은 특별한 일은 아니다. 집계할 데이터는 고정 빈도를 가질 필요가 없으며 잘라낸 시계열 조각의 크기를 원하는 빈도로 정의한다. 예를 들어 'M', 'BM' 같은 월간 빈도로 변환하려면 데이터를 월 간격으로 나눠야 한다. 각 간격은 한쪽이 열려 있는데. 이 말은 하나의 간격에서 양끝 중 한쪽만 포함된다는 뜻이다. 그러면 각 간격의 모음이 전체 시계열이 된다. resample을 사용해서 데이터를 다운샘플링할 때 고려해야 할 사항이 몇 가지 있다.

- **각 간격의 양끝 중에서 어느 쪽을 닫아둘 것인가**
- **집계하려는 구간의 라벨을 간격의 시작으로 할지 끝으로 할지 여부**

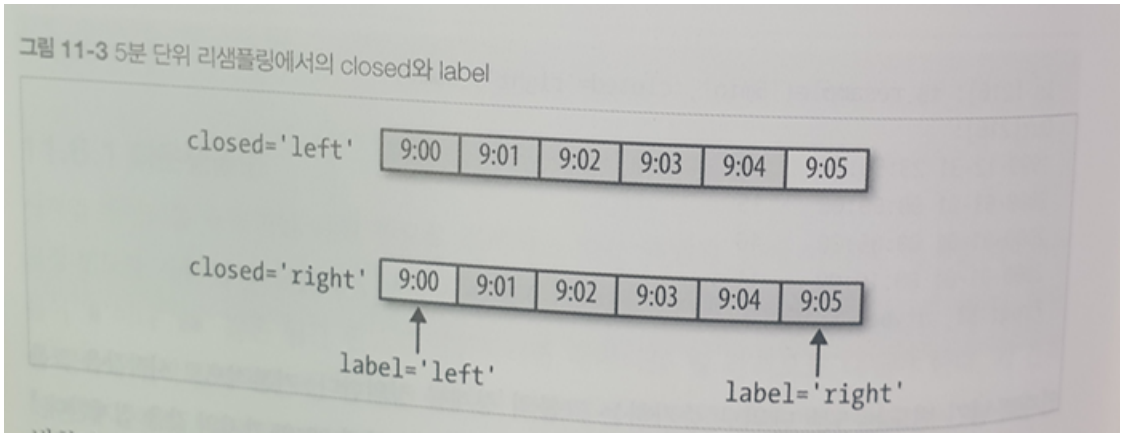
분 단위 데이터를 통해 좀 더 알아보자.

```
In [214]: rng = pd.date_range('2000-01-01',periods = 12, freq = 'T')

In [215]: ts = pd.Series(np.arange(len(rng)),index = rng)

In [216]: ts
Out[216]:
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: int32
```

이 데이터를 5분 단위로 묶어서 각 그룹의 합을 집계해보자.

```
In [217]: ts.resample('5min', closed = 'right').sum()
Out[217]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int32

In [218]: ts.resample('5min', closed = 'left').sum()
Out[218]:
2000-01-01 00:00:00    10
2000-01-01 00:05:00    35
2000-01-01 00:10:00    21
Freq: 5T, dtype: int32
```

인자로 넘긴 빈도는 5분 단위로 증가하는 그룹의 경계를 정의한다. 기본적으로 시작값을 그룹의 **왼쪽**에 포함시키므로 00:00의 값은 첫 번째 그룹의 00:00~00:05 까지의 값을 집계한다.



closed = 'right' 를 넘기면 시작값을 그룹의 오른쪽에 포함시킨다. 결과로 반환된 시계열은 각 그룹의 왼쪽 타임스탬프가 라벨로 사용할 수 있다.

반환된 결과의 색인을 특정 크기만큼 이동시키고 싶은 경우, 즉 그룹의 오른쪽 끝에서 1초를 빼서 타임스탬프가 참조하는 간격을 좀 더 명확히 보여주고 싶은 경우에는 loffset 메서드에 문자열이나 날짜 오프셋을 넘기면 된다.

```
In [220]: ts.resample('5min',closed = 'right', label = 'right',loffset=
     ...:  '-1s').sum()
Out[220]:
1999-12-31 23:59:59     0
2000-01-01 00:04:59    15
2000-01-01 00:09:59    40
2000-01-01 00:14:59    11
Freq: 5T, dtype: int32
```

loffset 대신 반환된 결과에 shift 메서드를 사용해도 같은 결과를 얻을 수 있다.

OHLC 리샘플링

금융 분야에서 시계열 데이터를 집계하는 아주 흔한 방식은 각 버킷에 대해 4가지 값을 계산하는 것이다. 이 4가지 값은 시가, 고가, 저가, 종가 이며 이를 OHLC(Open-High-Low-Close) 라고 한다.

how = 'ohlc'를 넘겨서 한 번에 이 값을 담고 있는 컬럼을 가지고 DataFrame을 얻을 수 있다.

```
In [221]: ts.resample('5min').ohlc()
Out[221]:
                     open  high  low  close
2000-01-01 00:00:00     0     4    0      4
2000-01-01 00:05:00     5     9    5      9
2000-01-01 00:10:00    10    11   10     11
```

11.6.2 업샘플링과 보간

하위 빈도에서 상위 빈도로 변환할 때는 집계가 필요하지 않다. 주간 데이터를 담고 있는 DataFrame을 살펴보자.

```
In [224]: 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'])

In [225]: frame
Out[225]:
            Colorado     Texas  New York      Ohio
2000-01-05  0.223289  0.172885  0.140301  1.072137
2000-01-12  0.005513 -0.296906 -0.594491 -2.525603
```

이 데이터에 요약함수를 사용하면 그룹당 하나의 값이 들어가고 그 사이에 결측치가 들어간다. asfreq 메서드를 이용해서 어떤 요약함수도 사용하지 않고 상위 빈도로 리샘플링해보자.

```
In [227]: df_daily = frame.resample('D').asfreq()

In [228]: df_daily
Out[228]:
            Colorado     Texas  New York      Ohio
2000-01-05  0.223289  0.172885  0.140301  1.072137
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  0.005513 -0.296906 -0.594491 -2.525603
```

수요일이 아닌 요일에는 이전 값을 채워서 보간을 수행한다고 가정하자. fillna와 reindex 메서드에서 사용했던 보간 메서드를 리샘플링에서도 사용할 수 있다.

```
In [229]: frame.resample('D').ffill()
Out[229]:
            Colorado     Texas  New York      Ohio
2000-01-05  0.223289  0.172885  0.140301  1.072137
2000-01-06  0.223289  0.172885  0.140301  1.072137
2000-01-07  0.223289  0.172885  0.140301  1.072137
2000-01-08  0.223289  0.172885  0.140301  1.072137
2000-01-09  0.223289  0.172885  0.140301  1.072137
2000-01-10  0.223289  0.172885  0.140301  1.072137
2000-01-11  0.223289  0.172885  0.140301  1.072137
2000-01-12  0.005513 -0.296906 -0.594491 -2.525603
```

limit 옵션을 사용해서 보간법을 적용할 범위를 지정하는 것도 가능하다.

```
In [230]: frame.resample('D').ffill(limit = 2)
Out[230]:
            Colorado     Texas  New York      Ohio
2000-01-05  0.223289  0.172885  0.140301  1.072137
2000-01-06  0.223289  0.172885  0.140301  1.072137
2000-01-07  0.223289  0.172885  0.140301  1.072137
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  0.005513 -0.296906 -0.594491 -2.525603
```

특히 새로운 날짜 색인은 이전 색인과 겹쳐질 필요가 전혀 없다.

```
In [231]: frame.resample('W-THU').ffill()
Out[231]:
            Colorado     Texas  New York      Ohio
2000-01-06  0.223289  0.172885  0.140301  1.072137
2000-01-13  0.005513 -0.296906 -0.594491 -2.525603
```

11.6.3 기간 리샘플링

기간으로 색인된 데이터를 리샘플링하는 것은 타임스탬프와 유사하다.

```
In [234]: frame = pd.DataFrame(np.random.rand(24,4),
     ...:                      index = pd.period_range('1-2000','12-20
     ...: 01',freq='M'),columns = ['Colorado', 'Texas', 'New York','Ohi
     ...: o'])

In [235]: frame[:5]
Out[235]:
         Colorado     Texas  New York      Ohio
2000-01  0.444994  0.037958  0.007944  0.912905
2000-02  0.235990  0.924889  0.911558  0.644603
2000-03  0.823274  0.048648  0.825722  0.411822
2000-04  0.609663  0.402892  0.667382  0.444334
2000-05  0.620886  0.351928  0.521427  0.551831

In [236]: annual_frame = frame.resample('A-DEC').mean()

In [237]: annual_frame
Out[237]:
      Colorado     Texas  New York      Ohio
2000  0.552851  0.401923  0.623274  0.626321
2001  0.523261  0.472095  0.401737  0.537062
```

업샘플링은 asfreq 메서드처럼 리샘플링하기 전에 새로운 빈도에서 구간의 끝을 어느 쪽에 두어야 할지 미리 결정해야 한다. convention 인자의 기본값은 'start'지만 'end'지정할 수 도 있다.

```
In [238]: annual_frame.resample('Q-DEC').ffill()
Out[238]:
        Colorado     Texas  New York      Ohio
2000Q1  0.552851  0.401923  0.623274  0.626321
2000Q2  0.552851  0.401923  0.623274  0.626321
```

```
2000Q3  0.552851  0.401923  0.623274  0.626321
2000Q4  0.552851  0.401923  0.623274  0.626321
2001Q1  0.523261  0.472095  0.401737  0.537062
2001Q2  0.523261  0.472095  0.401737  0.537062
2001Q3  0.523261  0.472095  0.401737  0.537062
2001Q4  0.523261  0.472095  0.401737  0.537062

In [239]: annual_frame.resample('Q-DEC', convention='end').ffill()
Out[239]:
        Colorado     Texas  New York      Ohio
2000Q4  0.552851  0.401923  0.623274  0.626321
2001Q1  0.552851  0.401923  0.623274  0.626321
2001Q2  0.552851  0.401923  0.623274  0.626321
2001Q3  0.552851  0.401923  0.623274  0.626321
2001Q4  0.523261  0.472095  0.401737  0.537062
```

기간의 업샘플링과 다운샘플링은 좀 더 엄격하다.

- 다운샘플링의 경우 대상 빈도는 반드시 원본 빈도의 **하위 기간**이어야 한다.
- 업샘플링의 경우 대상 빈도는 반드시 원본 빈도의 **상위 기간**이어야 한다.

위 조건을 만족하지 않으면 예외가 발생한다. 이 예외는 주로 분기, 연간, 주간 빈도에서 발생하는데, 예를 들어 Q-MAR로 정의된 기간은 A-MAR, A-JUN, A-SEP, A-DEC로만 이뤄져 있다.

```
In [240]: annual_frame.resample('Q-MAR').ffill()
Out[240]:
        Colorado     Texas  New York      Ohio
2000Q4  0.552851  0.401923  0.623274  0.626321
2001Q1  0.552851  0.401923  0.623274  0.626321
2001Q2  0.552851  0.401923  0.623274  0.626321
2001Q3  0.552851  0.401923  0.623274  0.626321
2001Q4  0.523261  0.472095  0.401737  0.537062
2002Q1  0.523261  0.472095  0.401737  0.537062
2002Q2  0.523261  0.472095  0.401737  0.537062
2002Q3  0.523261  0.472095  0.401737  0.537062

In [246]: annual_frame.resample('Q-SEP').ffill()
Out[246]:
        Colorado     Texas  New York      Ohio
2000Q2  0.552851  0.401923  0.623274  0.626321
2000Q3  0.552851  0.401923  0.623274  0.626321
2000Q4  0.552851  0.401923  0.623274  0.626321
2001Q1  0.552851  0.401923  0.623274  0.626321
2001Q2  0.523261  0.472095  0.401737  0.537062
2001Q3  0.523261  0.472095  0.401737  0.537062
2001Q4  0.523261  0.472095  0.401737  0.537062
2002Q1  0.523261  0.472095  0.401737  0.537062
```

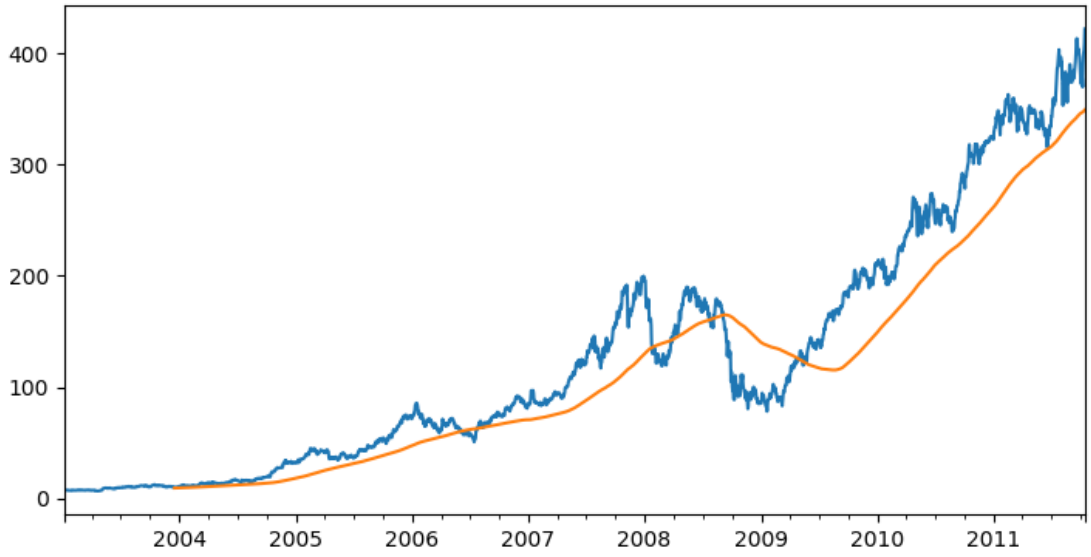
11.7 이동창 함수

시계열 연산에서 사용되는 배열 변환에서 중요한 요소는 움직이는 창 또는 지수 가중과 함께 수행되는 통계와 여타 함수들이다. 이런 함수를 이용해서 누락된 데이터로 인해 매끄럽지 않은 시계열 데이터를 매끄럽게 다듬을 수 있다. 나는 지수 가중 이동평균 처럼 고정 크기의 창을 가지지 않는 함수도

포함해서 **이동창 함수**라고 부른다. 다른 통계 함수와 마찬가지로 이동창 함수도 누락된 데이터를 자동으로 배제한다.

```
In [251]: close_px_all = pd.read_csv('stock_px_2.csv',parse_dates=True,
     ...: index_col = 0)

In [252]: close_px_all.head()
Out[252]:
            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

In [253]: close_px = close_px_all[['AAPL','MSFT','XOM']]

In [261]: close_px = close_px.resample('B').ffill()

In [262]: close_px
Out[262]:
              AAPL   MSFT    XOM
2003-01-02    7.40  21.11  29.22
2003-01-03    7.45  21.14  29.24
2003-01-06    7.45  21.52  29.96
2003-01-07    7.43  21.93  28.95
2003-01-08    7.28  21.31  28.83
...            ...    ...    ...
2011-10-10  388.81  26.94  76.28
2011-10-11  400.29  27.00  76.27
2011-10-12  402.19  26.96  77.16
2011-10-13  408.43  27.18  76.37
2011-10-14  422.00  27.27  78.11
```

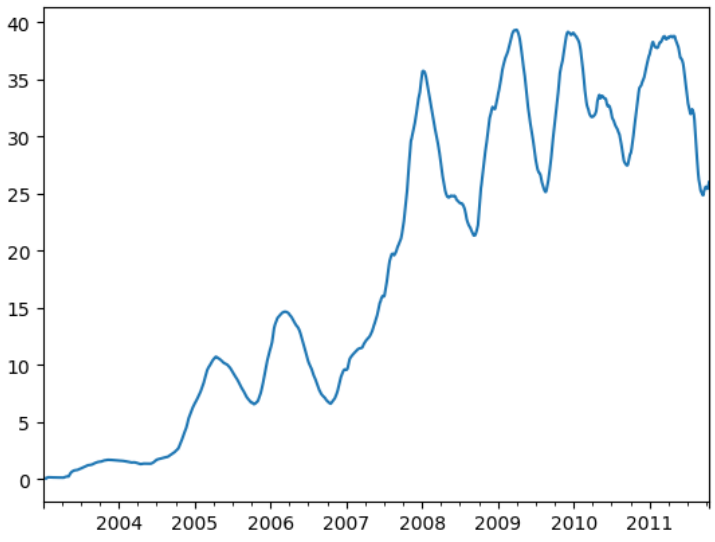
이제 resample 이나 groupby와 유사하게 작동하는 rolling 연산을 알아보자. 이는 Series나 DataFrame에 대해 원하는 기간을 나타내는 window 값과 함께 호출할 수 있다. 다음은 이 데이터를 시각화 한것이다.



rolling(250)이라는 표현은 groupby와 비슷해 보이지만 그룹을 생성하는 대신 250일 크기의 움직이는 창을 통해 그룹핑할 수 있는 객체를 생성한다. 다음은 250일 일 별 수익 표준편차를 나타낸 그래프다.

rolling 함수는 기본적으로 해당 윈도우 내에는 결측치가 없기를 기대하지만 시계열의 시작 지점에서는 필연적으로 window보다 적은 기간의 데이터를 가지고 있으므로 이를 처리하기 위해 rolling 함수의 동작 방식은 변경 할 수 있다.

```
In [270]: appl_std250 = close_px.AAPL.rolling(250, min_periods = 10).st
     ...: d()

In [271]: appl_std250
Out[271]:
2003-01-02          NaN
2003-01-03          NaN
2003-01-06          NaN
2003-01-07          NaN
2003-01-08          NaN
                 ...
2011-10-10    25.430104
2011-10-11    25.523131
2011-10-12    25.624851
2011-10-13    25.758644
2011-10-14    25.993449
Freq: B, Name: AAPL, Length: 2292, dtype: float64

In [272]: appl_std250[10:20]
Out[272]:
2003-01-16    0.074760
2003-01-17    0.112368
2003-01-20    0.132011
2003-01-21    0.149286
2003-01-22    0.169008
2003-01-23    0.168760
2003-01-24    0.184135
2003-01-27    0.182546
2003-01-28    0.178117
2003-01-29    0.181870
Freq: B, Name: AAPL, dtype: float64

In [273]: appl_std250.plot()
Out[273]: <matplotlib.axes._subplots.AxesSubplot at 0x1c53b5b7b88>
```

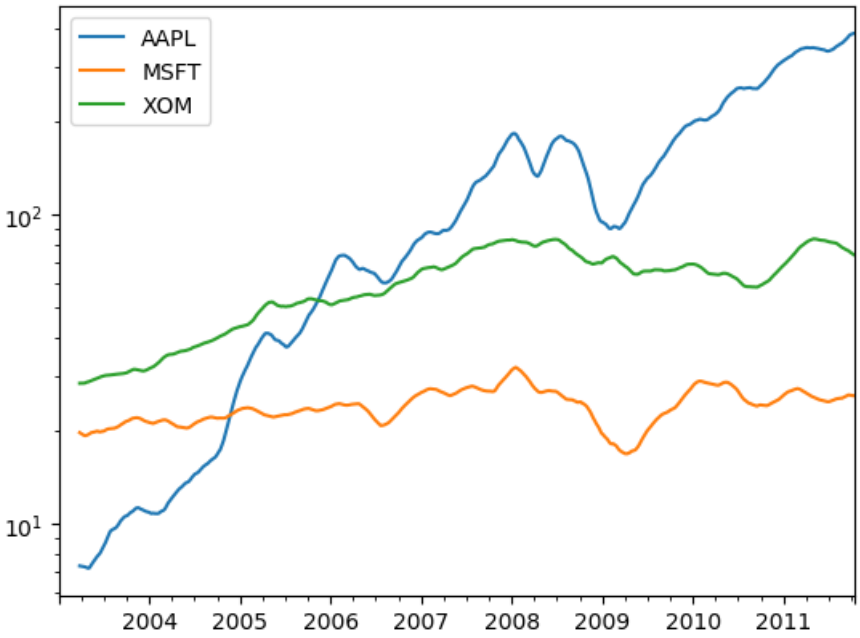


확장창 평균을 구하기 위해서는 rolling 대신 expanding을 사용한다. 확장창 평균은 시계열의 시작 지점에서부터 창의 크기가 시계열의 전체 크기가 될 때 까지 점점 창의 크기를 늘린다.

```
In [275]: expanding_mean = appl_std250.expanding().mean()
```

DataFrame 에 대해 이동창 함수를 호출하면 각 컬럼에 적용된다.

```
In [276]: close_px
Out[276]:
               AAPL   MSFT    XOM
2003-01-02     7.40  21.11  29.22
2003-01-03     7.45  21.14  29.24
2003-01-06     7.45  21.52  29.96
2003-01-07     7.43  21.93  28.95
2003-01-08     7.28  21.31  28.83
...             ...    ...    ...
2011-10-10  388.81  26.94  76.28
2011-10-11  400.29  27.00  76.27
2011-10-12  402.19  26.96  77.16
2011-10-13  408.43  27.18  76.37
2011-10-14  422.00  27.27  78.11

[2292 rows x 3 columns]

In [277]: close_px.rolling(60).mean().plot(logy=True)
Out[277]: <matplotlib.axes._subplots.AxesSubplot at 0x1c539c834c8>
```



rolling 함수는 고정 크기의 기간 지정 문자열을 넘겨서 호출할 수도 있다. 빈도가 불규칙한 시계열일 경우 유용하게 사용할 수 있다. resample 함수에서 사용하던 것과 같은 형식이다. 예를 들어 20일 크기의 이동 평균은 아래처럼 구할 수 있다.

```
In [281]: close_px.rolling('20D').mean()
Out[281]:
                 AAPL        MSFT         XOM
2003-01-02   7.400000   21.110000   29.220000
```

```
2003-01-03    7.425000  21.125000  29.230000
2003-01-06    7.433333  21.256667  29.473333
2003-01-07    7.432500  21.425000  29.342500
2003-01-08    7.402000  21.402000  29.240000
...                ...        ...        ...
2011-10-10  389.351429  25.602143  72.527857
2011-10-11  388.505000  25.674286  72.835000
2011-10-12  388.531429  25.810000  73.400714
2011-10-13  388.826429  25.961429  73.905000
2011-10-14  391.038000  26.048667  74.185333

[2292 rows x 3 columns]
```

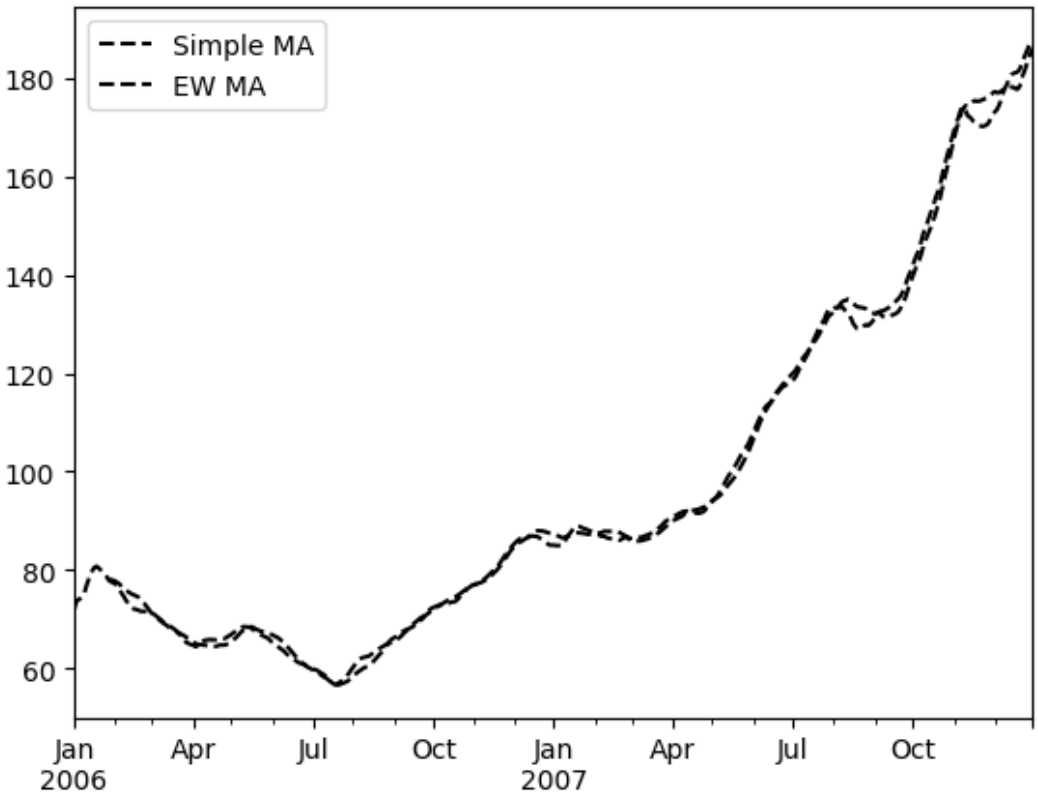
11.7.1 지수 가중 함수

균등한 가중치를 가지는 관찰과 함께 고정 크기 창을 사용하는 다른 방법은 **감쇠인자** 상수에 좀 더 많은 가중치를 줘서 더 최근 값을 관찰하는 것이다. 감쇠인자 상수를 지정하는 방법은 몇 가지 있는데 널리 쓰는 방법은 기간을 이용하는 것이다. 이 방법은 결과를 같은 기간의 창을 가지는 단순 이동창 함수와 비교 가능하도록 해준다.

지수 가중 통계는 최근 값에 좀 더 많은 가중치를 두는 방법이므로 균등 가중 방식에 비해 좀 더 빠르게 변화를 수용한다.

pandas는 rolling 이나 expanding 과 함께 사용할 수 있는 ewm 연산을 제공한다. 아래 예제는 애플 주가 60일 이동평균을 span=60 으로 구한 지수 가중 이동평균과 비교한 것이다.

```
In [284]: aapl_px = close_px.AAPL['2006':'2007']

In [285]: ma60 = aapl_px.rolling(30,min_periods = 20).mean()

In [286]: ewma60 = aapl_px.ewm(span = 30).mean()

In [287]: ma60.plot(style = 'k--', label='Simple MA')
Out[287]: <matplotlib.axes._subplots.AxesSubplot at 0x1c543fb2048>

In [290]: ewma60.plot(style = 'k--', label = 'EW MA')
Out[290]: <matplotlib.axes._subplots.AxesSubplot at 0x1c5440f32c8>

In [291]: plt.legend()
Out[291]: <matplotlib.legend.Legend at 0x1c53ae4ca48>
```



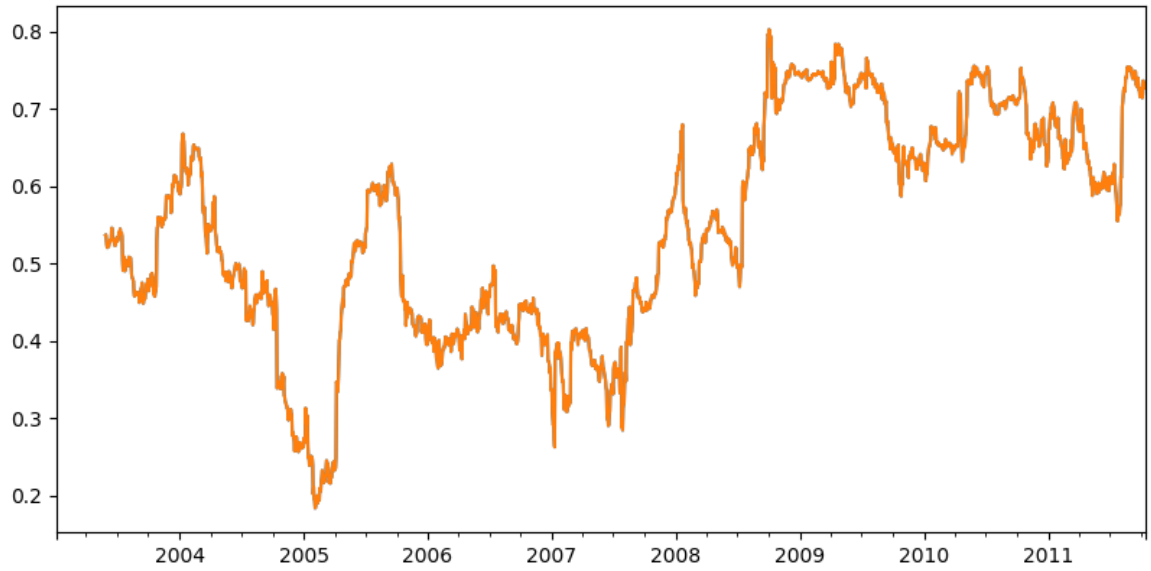
11.7.2 이진 이동창 함수

상관관계와 공분산 같은 몇몇 통계 연산은 두 개의 시계열을 필요로 한다. 예를 들어보자. 금융 분석가는 종종 S&P500 같은 비교 대상이 되는 지수와 주식의 상관관계에 흥미를 가진다.

```
In [8]: spx_px = close_px_all['SPX']

In [9]: spx_rets = spx_px.pct_change()

In [10]: close_px = close_px_all[['AAPL','MSFT','XOM']]

In [11]:  close_px = close_px.resample('B').ffill()

In [12]: returns = close_px.pct_change()
```

rolling 함수에 이어 호출한 corr 요약함수는 spx_rets와의 상관관계를 계산한다.

```
In [36]: corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_ret
    ...: s)

In [37]: corr.plot()
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x1a0a36dcf08>
```



여러 주식과 S&P 500 지수와의 상관관계를 한번에 계산하고 싶다고 가정하자. 반복문을 작성해서 새로운 DataFrame을 생성하면 쉽겠지만 좋은 방법은 아니다. TimeSerise와 DataFrame 그리고 rolling_corr 같은 함수를 넘겨서 TimeSeries(이 경우에는 spx_rets) 와 DataFrame의 각 컬럼 간의 상관관계를 계산하면 된다.

```
In [38]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)

In [39]: corr.plot()
Out[39]: <matplotlib.axes._subplots.AxesSubplot at 0x1a0a3af2e88>
```

11.7.3 사용자 정의 이동창 함수

rolling 이나 다른 관련 메서드에 apply를 호출해서 이동창에 대한 사용자 정의 연산을 수행할 수 있다. 유일한 요구 사항은 사용자 정의 함수가 배열의 각 조각으로부터 단일 값(감소)을 반환해야 한다는 것이다. 예를 들어 rolling(...), quantile(q)를 사용해서 표본 변위치를 계산할 수 있는 것처럼 전체 표본에서 특정 값이 차지하는 백분위 점수를 구하는 함수를 작성할 수도 있다.

scipy.stats.percentileofscore 함수가 그런 기능을 한다.

```
In [41]: from scipy.stats import percentileofscore

In [42]: score_at_2percent = lambda x : percentileofscore(x,0.02)

In [43]: result = returns.AAPL.rolling(250).apply(score_at_2percent)

In [44]: result.plot()
Out[44]: <pandas.plotting._core.PlotAccessor object at 0x000001A0ABD14DC8>
```