8. 데이터 준비하기 : 조인, 병합, 변형

대부분의 경우 데이터는 여러 파일이나 DB혹은 분서하기 쉽지 않은 형태로 기록되어 있다. 이 장에서는 데이터를 합치고, 재배열할 수 있는 도구들을 살펴보자.

먼저 데이터를 병합하거나 변환하는 과정에서 사용되는 pandas의 계측적 색인의 개념을 알아보고 이를 활용하여 데이터를 다듬는 과정을 심도 있게 살펴볼 것이다. 14장에서도 이런 도구를 사용하는 다양한 예시를 볼 수 있다.

8.1 계층적 색인

계층적 색인은 pandas의 중요한 기능인데, 축에 대해 다중(둘 이상) 색인 단계를 지정할 수 있도록 해준다. 약간 추상적으로 말하면, 높은 차원의 데이터를 낮은 차원의 형식으로 다룰 수 있게 해주는 기능이다. 간단한 예제를 하나 살펴보자. 우선 리스트의 리스트(또는 배열)를 색인으로 하는 Series를 하나 생성하자.

```
In [133]: data = pd.Series(np.random.randn(9),
     ...:                  index = [['a','a','a','b','b','c','c','d','d'],
     ...:                  [1,2,3,1,3,1,2,2,3]])

In [134]: data
Out[134]:
a  1    2.215759
   2    0.732009
   3   -1.120369
b  1    0.190680
   3   -0.359811
c  1    0.952093
   2   -0.273781
d  2    0.846637
   3    2.288906
dtype: float64
```

지금 생성한 객체가 MultiIndex를 색인으로 하는 Series인데, 색인의 계층을 보여주고 있다.

바로 위 단계의 색인을 이용해서 하위 계층을 직접 접근할 수 있다.

```
In [135]: data.index
Out[135]:
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

계층적으로 색인된 객체는 데이터의 부분집합을 **부분적 색인으로 접근** 하는 것이 가능하다.

```
In [136]: data['b']
```

```
Out[136]:
1    0.190680
3   -0.359811
dtype: float64

In [137]: data['b':'c']
Out[137]:
b  1    0.190680
   3   -0.359811
c  1    0.952093
   2   -0.273781
dtype: float64

In [139]: data.loc[['b','d']]
Out[139]:
b  1    0.190680
   3   -0.359811
d  2    0.846637
   3    2.288906
dtype: float64
```

하위 계층의 객체를 선책하는 것도 가능하다.

```
In [140]: data.loc[:,2]
Out[140]:
a    0.732009
c   -0.273781
d    0.846637
dtype: float64
```

계층적인 색인은 데이터를 재형성하고 피벗테이블 생성 같은 그룹 기반의 작업을 할 때 중요하게 사용된다. 예를 들어 위에서 만든 DataFrame 객체에 unstack 메서드를 사용해서 데이터를 새롭게 배열할 수도 있다.

```
In [141]: data.unstack()
Out[141]:
          1         2         3
a  2.215759  0.732009 -1.120369
b  0.190680       NaN -0.359811
c  0.952093 -0.273781       NaN
d       NaN  0.846637  2.288906
```

unstack의 반대작업은 stack 메서드로 수행한다.

```
In [142]: data.unstack().stack()
Out[142]:
a  1    2.215759
   2    0.732009
   3   -1.120369
b  1    0.190680
   3   -0.359811
c  1    0.952093
   2   -0.273781
d  2    0.846637
   3    2.288906
dtype: float64
```

stack과 unstack 메서드는 이 장 후반부에서 더 자세히 알아보기로 하자.

DataFrame에서는 두 축 모두 계층적 색인을 가질 수 있다.

```
In [144]: frame = pd.DataFrame(np.arange(12).reshape((4,3)),
     ...:                      index=[['a','a','b','b'],
     ...:                             [1,2,1,2]],
     ...:                      columns = [['Ohio','Ohio','Colorado'],
     ...:                                 ['Green','Red','Green']])

In [145]: frame
Out[145]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

계층적 색인의 각 단계는 이름(문자열이나 어떤 파이썬 객체라도 가능하다)을 가질 수 있고, 만약 이름을 가지고 있다면 콘솔 출력 시 함께 나타난다.

```
In [146]: frame.index.names = ['key1','key2']

In [147]: frame.columns.names = ['state','color']

In [148]: frame
Out[148]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

CAUTION_ 색인 이름인 'state'와 'color'를 로우 라벨과 혼동하지 말자.

칼럼의 부분집합을 부분적인 색인으로 접근하는 것도 컬럼에 대한 부분적 색인과 비슷하게 사용하면 된다.

```
In [149]: frame['Ohio']
Out[149]:
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

MultiIndex는 따로 생성한 다음에 재사용 가능하다. 위에서 살펴본 DataFrame의 컬럼 계층 이름은 다음처럼 생성할 수 있다.

```
In [152]: MultiIndex.from_arrays([['Ohio','Ohio','Colorado'],
     ...:                         ['Green','Red','Green']],
     ...:                        names = ['state','color'] )
```

8.1.1 계층의 순서를 바꾸고 정렬하기

계층적 색인에서 계층의 순서를 바꾸거나 지정된 계층에 따라 데이터를 정렬해야 하는 경우가 있을 수 있다. swaplevel은 넘겨받은 두 개의 계층 번호나 이름이 뒤바뀐 새로운 객체를 반환한다(하지만 데이터는 변경되지 않는다).

```
In [153]: frame.swaplevel('key1','key2')
Out[153]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
2    a        3   4        5
1    b        6   7        8
2    b        9  10       11
```

반면 sort_index 메서드는 단일 계층에 속한 데이터를 정렬한다. swaplevel을 이용해서 계층을 바꿀 때 sort_index를 사용해서 결과가 사전적으로 정렬되도록 만드는 것도 드물지 않은 일이다.

```
In [154]: frame.sort_index(level=1)
Out[154]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
b    1        6   7        8
a    2        3   4        5
b    2        9  10       11

In [155]: frame.sort_index(level=0)
Out[155]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

Note_ 객체가 계층적 색인으로 상위 계층부터 사전식으로 정렬되어 있다면 (sort_index(level=0) 이나 sort_index()의 결과처럼) 데이터를 선택하는 성능이 훨씬 좋아진다.

8.1.2 계층별 요약 통계

DataFrame과 Series의 많은 기술 통계와 요약 통계는 level 옵션을 가지고 있는데, 어떤 한 축에 대해 합을 구하고 싶은 단계를 지정할 수 있는 옵션이다. 앞에서 살펴본 DataFrame에서 로우나 컬럼을 아래처럼 계층별로 합칠 수 있다.

```
In [8]: frame.sum(level='key2')
Out[8]:
state  Ohio     Colorado
color Green Red    Green
key2
1         6   8       10
2        12  14       16

In [9]: frame.sum(level='color',axis = 1)
Out[9]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

이는 내부적으로 pandas의 groupby 기능을 이용해서 구현했다.

8.1.3 DataFrame의 컬럼 사용하기

DataFrame에서 로우를 선택하기 위한 색인이나 하나 이상의 컬럼을 사용하는 것은 드물지 않은 일이다.

```
In [10]: frame = pd.DataFrame({'a':range(7),
    ...:                       'b':range(7,0,-1),
    ...:                       'c':['one','one','one','two'
    ...:                       ,'two','two','two'],
    ...:                       'd':[0,1,2,0,1,2,3]})

In [11]: frame
Out[11]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

DataFrame의 set_index 함수는 하나 이상의 컬럼을 색인으로 하는 새로운 DataFrame을 생성한다.

```
In [12]: frame2 = frame.set_index(['c','d'])

In [13]: frame2
Out[13]:
       a  b
c   d
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

다음처럼 컬럼을 명시적으로 남겨두지 않으면 DataFrame에서 삭제된다.

```
In [14]: frame.set_index(['c','d'],drop=False)
Out[14]:
       a  b    c  d
c   d
one 0  0  7  one  0
    1  1  6  one  1
    2  2  5  one  2
two 0  3  4  two  0
    1  4  3  two  1
    2  5  2  two  2
    3  6  1  two  3
```

반면 reset_index 함수는 set_index와 반대되는 개념인데 계층적 색인 단계가 컬럼으로 이동한다.

```
In [15]: frame2.reset_index()
Out[15]:
     c  d  a  b
0  one  0  0  7
1  one  1  1  6
2  one  2  2  5
3  two  0  3  4
4  two  1  4  3
5  two  2  5  2
6  two  3  6  1
```

8.2 데이터 합치기

pandas 객체에 저장된 데이터는 여러 가지 방법으로 합칠 수 있다.

- **pandas.merge는 하나 이상의 키를 기준으로 DataFrame의 로우와 합친다. SQL이나 다른 관계형 DB의 join연산과 유사하다.**
- **pandas.concat은 하나의 축을 따라 객체를 이어붙인다.**
- **combine_first 인스턴스 메서드는 두 객체를 포개서 한 객체에서 누락된 데이터를 다른 객체에 있는 값으로 채울 수 있도록 한다.**

각각의 데이터를 합치는 방법에 대해 다양한 예제와 함께 살펴보게 될 것이다.

8.2.1. DB스타일로 DataFrame 합치기

병합이나 **조인**연산은 관계형 DB의 핵심적인 연산인데, 하나 이상의 **키**를 사용해서 데이터 집합의 로우를 합친다. pandas의 merge 함수를 이용해서 이런 알고리즘을 데이터에 적용할 수 있다.

예제를 보자.

```
In [16]: df1 = pd.DataFrame({'key':['b','b','a','c','a','a','b'],
    ...:                     'data1':range(7)})

In [17]: df1
Out[17]:
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   a      5
6   b      6

In [27]: df2 = pd.DataFrame({'key':['a','b','d'],
    ...:                     'data2':range(3)})

In [28]: df2
Out[28]:
  key  data2
0   a      0
1   b      1
2   d      2
```

위 예제는 **다대일**의 경우다. df1의 데이터는 key 컬럼에 여러 개의 a,b 로우를 가지고 있고 df2의 데이터는 key 컬럼에 유일한 로우를 가지고 있다. 이 객체에 대해 merge 함수를 호출하면 다음과 같다.

```
In [29]: pd.merge(df1,df2)
Out[29]:
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
```

위에서 나는 어떤 컬럼을 병합할 것인지 명시하지 않았는데, merge 함수는 중복된 컬럼이름을 키로 사용한다.

```
In [30]: pd.merge(df1,df2,on='key')
Out[30]:
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
```

만약 두 객체에 중복된 컬럼이름이 하나도 없다면 따로 지정해주면된다.

```
In [31]: df3 = pd.DataFrame({'lkey':['b','b','a','c','a','a','b'],
    ...:                     'data1':range(7)})

In [32]: df4 = pd.DataFrame({'rkey':['a','b','d'],
    ...:                     'data2':range(3)})

In [36]: df3
Out[36]:
  lkey  data1
0    b      0
1    b      1
2    a      2
3    c      3
4    a      4
5    a      5
6    b      6

In [37]: df4
Out[37]:
  rkey  data2
0    a      0
1    b      1
2    d      2
```

```
In [34]: pd.merge(df3,df4,left_on='lkey',right_on='rkey')
Out[34]:
  lkey  data1 rkey  data2
0    b      0    b      1
1    b      1    b      1
2    b      6    b      1
3    a      2    a      0
4    a      4    a      0
5    a      5    a      0
```

결과를 잘 살펴보면 'c'와 'd'에 해당하는 값이 빠진 값을 알 수 있다. merge 함수는 기본적으로 내부 조인을 수행하여 교집합인 결과를 반환한다. how 인자로 'left','right','outer' 를 넘겨서 각각 왼쪽 조인, 오른쪽 조인, 외부 조인을 수행할 수도 있다. 외부 조인은 합집합인 결과를 반환하고 왼쪽 조인과 오른쪽 조인은 각각 왼쪽 또는 오른쪽의 모든 로우를 포함하는 결과를 반환한다.

다대다병합은 잘 정의되어 있긴 하지만 직관적이지는 않다.

```
In [40]: df1 = pd.DataFrame({'key1':['b','b','a','c','a','b'],
    ...:                     'data1':range(6)})

In [41]: df2 = pd.DataFrame({'key1':['a','b','a','b','d'],
    ...:                     'data1':range(5)})

In [42]: df1
Out[42]:
  key1  data1
0    b      0
1    b      1
2    a      2
3    c      3
```

```
4    a      4
5    b      5

In [43]: df2
Out[43]:
  key1  data1
0    a      0
1    b      1
2    a      2
3    b      3
4    d      4

In [46]: pd.merge(df1,df2,on='key1',how='left')
Out[46]:
   key1  data1_x  data1_y
0     b        0      1.0
1     b        0      3.0
2     b        1      1.0
3     b        1      3.0
4     a        2      0.0
5     a        2      2.0
6     c        3      NaN
7     a        4      0.0
8     a        4      2.0
9     b        5      1.0
10    b        5      3.0
```

다대다 조인은 두 로우의 데카르트곱을 반환한다. 왼쪽 DataFrame에는 3개의 'b' 로우가 있고 오른쪽에는 2개의 'b'로우가 있으며, 결과는 6개의 'b'로우가 된다. 조인 메서드는 결과에 나타나듯이, 구별되는 키에 대해서만 적용된다.

```
In [50]: pd.merge(df1,df2,on='key1',how='inner')
Out[50]:
  key1  data1_x  data1_y
0    b        0        1
1    b        0        3
2    b        1        1
3    b        1        3
4    b        5        1
5    b        5        3
6    a        2        0
7    a        2        2
8    a        4        0
9    a        4        2
```

여러 개의 키를 병합하려면 이름이 담긴 리스트를 넘기면 된다.

```
In [51]: left = pd.DataFrame({'key1':['foo','foo','bar'],
    ...:                      'key2':['one','two','one'],
    ...:                      'key3':[1,2,3]})

In [52]: right = pd.DataFrame({'key1':['foo','foo','bar','bar'],
    ...:                       'key2':['one','one','one','two'],
    ...:                       'key3':[4,5,6,7]})

In [53]: pd.merge(left,right,on=['key1','key2'],how='outer')
```

```
Out[53]:
  key1 key2  key3_x  key3_y
0  foo  one     1.0     4.0
1  foo  one     1.0     5.0
2  foo  two     2.0     NaN
3  bar  one     3.0     6.0
4  bar  two     NaN     7.0

In [54]: left
Out[54]:
  key1 key2  key3
0  foo  one     1
1  foo  two     2
2  bar  one     3

In [55]: right
Out[55]:
  key1 key2  key3
0  foo  one     4
1  foo  one     5
2  bar  one     6
3  bar  two     7
```

merge 메서드의 종류에 따라 어떤 키 조합이 결과로 반환되는지 알려면 실제 구현과는 조금 다르지만 여러 개의 키가 들어 있는 튜플의 배열이 단일 조인키로 사용된다고 생각하면 된다.

CAUTION_ 칼럼과 컬럼을 조인할 때 전달한 DataFrame 객체의 색인은 무시된다.

병렬 연산에서 고려해야 할 마지막 사항은 겹치는 컬럼 이름에 대한 처리다. 다음에 살펴보겠지만 축 이름을 변경해서 수동으로 컬럼 이름이 겹치게 할 수도 있고, merge 함수에 있는 suffixes 인자로 두 DataFrame 객체에서 겹치는 컬럼 이름 뒤에 붙일 문자열을 지정해줄 수도 있다.

```
In [56]: pd.merge(left,right,on='key1')
Out[56]:
  key1 key2_x  key3_x key2_y  key3_y
0  foo    one       1    one       4
1  foo    one       1    one       5
2  foo    two       2    one       4
3  foo    two       2    one       5
4  bar    one       3    one       6
5  bar    one       3    two       7

In [57]: pd.merge(left, right, on='key1',suffixes =('_left','_right'))
Out[57]:
  key1 key2_left  key3_left key2_right  key3_right
0  foo       one          1        one           4
1  foo       one          1        one           5
2  foo       two          2        one           4
3  foo       two          2        one           5
4  bar       one          3        one           6
5  bar       one          3        two           7
```

다음은 merge 함수의 인자에 대한 정리이다.

인자	설명
left	병합하려는 DataFrame 중 왼쪽에 위치한 DataFrame
right	병합하려는 DataFrame 중 오른쪽에 위치한 DataFrame
how	조인 방법, 'inner','outer','left','right' 기본값은 'inner'
on	조인하려는 컬럼 이름. 반드시 두 DataFrame 객체 모두에 존재하는 이름이어야 한다. 만약 명시되지 않고 다른 조인키도 주어지지 않으면 left와 right에서 공통되는 컬럼을 조인키로 사용한다.
left_on	조인키로 사용할 left DataFrame의 컬럼
right_on	조인키로 사용할 right DataFrame의 컬럼
left_index	조인키로 사용할 left DataFrame의 색인 로우(다중 색인일 경우 키)
right_index	조인키로 사용할 right DataFrame의 색인 로우(다중 색인일 경우 키)
sort	조인키로 따라 병합된 데이터를 사전순으로 정렬. 기본값은 True, 대용량 데이터의 경우 False로 하면 성능상의 이득을
suffixes	칼럼 이름이 겹칠 경우, 각 컬럼 이름 뒤에 붙일 문자열의 튜플. 기본값은 ('_x','_y'). 만약 'data' 라는 칼럼 이름이 양쪽 DataFrame에 같이 존재하면 결과에서는 'data_x', 'data_y'로 보여진다.
copy	False일 경우, 예외적인 경우에 데이터가 결과로 복사되지 않도록 한다. 기본값은 항상 복사가 이뤄진다.
indicator	merge라는 이름의 특별한 컬럼을 추가하여 각 로우의 소스가 어디인지 나타낸다. 'left_only', 'right_only', 'both' 값을 가진다.

8.2.2 색인 병합하기

병합하려는 키가 **DataFrame의 색인일 경우**가 있다. 이런 경우에는 left_index = True 혹은 right_index=True 옵션(또는 둘 다)을 지정해서 해당 색인을 병합키로 사용할 수 있다.

```
In [61]: right1 = pd.DataFrame({'group_val':[3.5,7]},
    ...:                        index=['a','b'])

In [62]: left1 = pd.DataFrame({'key':['a','b','a','a','b','c'],
    ...:                       'value':range(6)})

In [63]: right1
Out[63]:
   group_val
a        3.5
b        7.0

In [64]: left1
Out[64]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
```

```
5   c      5

In [65]: pd.merge(left1, right1, left_on='key',right_index=True)
Out[65]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

계층 색인된 데이터는 암묵적으로 여러 키를 병합하는 것이라 약간 복잡하다.

```
In [77]: lefth = pd.DataFrame({'key1':['Ohio','Ohio','Ohio'
    ...:                                ,'Nevada','Nevada'],
    ...:                       'key2':[2000,2001,2002,2001,2002],
    ...:                       'data':np.arange(5.)})

In [78]: righth = pd.DataFrame(np.arange(12).reshape((6,2)),
    ...:                       index=[['Nevada','Nevada','Ohio','Ohio',
    ...:                               'Ohio','Ohio'],
    ...:                              [2001,2000,2000,2000,2001,2002]],
    ...:                       columns = ['event1','event2'] )

In [79]: lefth
Out[79]:
     key1  key2  data
0    Ohio  2000   0.0
1    Ohio  2001   1.0
2    Ohio  2002   2.0
3  Nevada  2001   3.0
4  Nevada  2002   4.0

In [80]: righth
Out[80]:
             event1  event2
Nevada 2001       0       1
       2000       2       3
Ohio   2000       4       5
       2000       6       7
       2001       8       9
       2002      10      11
```

양쪽에 공통적으로 존재하는 여러 개의 색인을 병합하는 것도 가능하다.

```
In [82]: left2 = pd.DataFrame([[1.,2.],[3.,4.],[5.,6.]],
    ...:                       index = ['a','c','e'],
    ...:                       columns = ['Ohio','Nevada'])

In [83]: right2 = pd.DataFrame([[7.,8.],[9.,10.],[11.,12.],[13.,14.]],
    ...:                       index = ['b','c','d','e'],
    ...:                       columns = ['Missouri','Alabama'])

In [84]: left2
Out[84]:
   Ohio  Nevada
```

```
a   1.0     2.0
c   3.0     4.0
e   5.0     6.0

In [85]: right2
Out[85]:
   Missouri  Alabama
b       7.0      8.0
c       9.0     10.0
d      11.0     12.0
e      13.0     14.0

In [86]: pd.merge(left2, right2, how='outer',left_index=True,
    ...:                          right_index=True)
Out[86]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

색인으로 병합할 때 DataFrame의 join 메서드를 사용하면 편리하다. join 메서드는 컬럼이 겹치지 않으며 완전히 같거나 유사한 색인 구조를 가진 여러 개의 DataFrame 객체를 병합할 때 사용할 수 있다. 위에서 본 예제는 다음처럼 작성할 수 있다.

```
In [89]: left2.join(right2, how='outer')
Out[89]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

과거에 작성된 pandas의 일부 코드 제약으로 인해 DataFrame의 join 메서드는 왼쪽 조인을 수행한다. join 메서드를 호출한 DataFrame의 컬럼 중 하나에 대한 조인을 수행하는 것도 가능하다.

```
In [91]: left1.join(right1, on='key')
Out[91]:
  key  value  group_val
0   a      0        3.5
1   b      1        7.0
2   a      2        3.5
3   a      3        3.5
4   b      4        7.0
5   c      5        NaN
```

마지막으로 색인 대 색인으로 두 DataFrame을 병합하려면 그냥 간단하게 병합하려는 DataFrame의 리스트를 join 메서드로 넘기면 된다. 하지만 보통 이런 병합은 아래에서 살펴볼 concat 메서드를 사용한다.

```
In [92]: another = pd.DataFrame([[7.,8.],[9.,10.],[11.,12.],[16.,17.]],
    ...:                        index = ['a','c','e','f'],
    ...:                        columns=['New York','Oregon'])
```

```
In [93]: another
Out[93]:
   New York  Oregon
a       7.0     8.0
c       9.0    10.0
e      11.0    12.0
f      16.0    17.0

In [94]: left2.join([right2,another])
Out[94]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
c   3.0     4.0       9.0     10.0       9.0    10.0
e   5.0     6.0      13.0     14.0      11.0    12.0

In [95]: left2.join([right2,another],how='outer')
Out[95]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
c   3.0     4.0       9.0     10.0       9.0    10.0
e   5.0     6.0      13.0     14.0      11.0    12.0
b   NaN     NaN       7.0      8.0       NaN     NaN
d   NaN     NaN      11.0     12.0       NaN     NaN
f   NaN     NaN       NaN      NaN      16.0    17.0
```

8.2.3 축 따라 이어붙이기

데이터를 합치는 또 다른 방법으로 이어붙이기가 있다. NumPy는 ndarray를 이어붙이는 concatenate 함수를 제공한다.

```
In [96]: arr = np.arange(12).reshape((3,4))

In [97]: arr
Out[97]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [98]: np.concatenate([arr,arr],axis=1)
Out[98]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

Series나 DataFrame 같은 pandas 객체의 컨텍스트 내부에는 축마다 이름이 있어서 배열을 쉽게 이어붙일 수 있도록 되어 있다. 이때 다음 사항을 고려해야 한다.

- **만약 연결하려는 두 객체의 색인이 서로 다르면 결과는 그 색인의 교집합이어야 하는가 아니면 합집합이어야 하는가?**
- **합쳐진 결과에서 합쳐지기전 객체의 데이터를 구분할 수 있어야 하는가?**
- **어떤 축으로 연결할 것인지 고려해야 하는가? 많은 경우 DataFrame의 기본 정수 라벨이 가장 먼저 무시된다.**

pandas의 concat 함수는 위 사항에 대한 답을 제공한다. concat 함수가 어떻게 동작하는지 다양한 예제로 알아보자. 색인이 겹치지 않는 3개의 Series 객체가 있다고 하자.

```
In [99]: s1 = pd.Series([0,1],index=['a','b'])

In [100]: s2 = pd.Series([2,3,4],index=['c','d','e'])

In [101]: s3 = pd.Series([5,6],index=['f','g'])

In [102]: s1
Out[102]:
a    0
b    1
dtype: int64

In [103]: s2
Out[103]:
c    2
d    3
e    4
dtype: int64

In [104]: s3
Out[104]:
f    5
g    6
dtype: int64
```

이 세 객체를 리스트로 묶어서 concat 함수에 전달하면 값과 색인을 연결해준다.

```
In [105]: pd.concat([s1,s2,s3])
Out[105]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

concat 함수는 axis =0 을 기본으로 하여 새로운 Series 객체를 생성한다. 만약 axis = 1을 넘긴다면 결과는 Series가 아니라 DataFrame이 될 것이다.

```
In [106]: pd.concat([s1,s2,s3], axis=1 )
Out[106]:
     0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

겹치는 축이 없기 때문에 외부 조인으로 정렬된 합집합을 얻었지만 join = 'inner'를 넘겨서 교집합을 구할 수도 있다.

```
In [107]: s4 = pd.concat([s1,s3])

In [108]: s4
Out[108]:
a    0
b    1
f    5
g    6
dtype: int64

In [109]: pd.concat([s1,s4],axis=1)
Out[109]:
     0  1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6

In [110]: pd.concat([s1,s4],axis=1,join='inner')
Out[110]:
   0  1
a  0  0
b  1  1
```

마지막 예제에서 'f'와 'g' 라벨은 join = 'inner' 옵션으로 인해 사라지게 된다.

join_axes 인자로 병합하려는 축을 직접 지정해줄 수도 있다.

Series를 이어붙이기 전의 개별 Series를 구분할 수 없는 문제가 생기는데, 이어붙인 축에 대해 계층적 색인을 생성하여 식별이 가능하도록 할 수 있다. 계층적 색인을 생성하려면 keys 인자를 사용하면 된다.

```
In [11]: result = pd.concat([s1,s1,s3],keys=['one','two','three'])

In [12]: result
Out[12]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64

In [13]: result.unstack()
Out[13]:
         a    b    f    g
one    0.0  1.0  NaN  NaN
two    0.0  1.0  NaN  NaN
three  NaN  NaN  5.0  6.0
```

Series를 axis = 1 로 병합할 경우 keys는 DataFrame의 컬럼 제목이 된다.

```
In [15]: pd.concat([s1,s2,s3],axis = 1 ,keys=['one','two','three'])
Out[15]:
   one  two  three
a  0.0  NaN    NaN
b  1.0  NaN    NaN
c  NaN  2.0    NaN
d  NaN  3.0    NaN
e  NaN  4.0    NaN
f  NaN  NaN    5.0
g  NaN  NaN    6.0
```

DataFrame 객체에 대해서도 지금까지와 같은 방식으로 적용할 수 있다.

```
In [16]: df1 = pd.DataFrame(np.arange(6).reshape((3,2)),index=['a','b','c'],
    ...:                    columns=['one','two'])
    ...:
    ...:
    ...:

In [17]: df2 = pd.DataFrame(5+np.arange(4).reshape(2,2),
    ...:                    index=['a','c'],
    ...:                    columns=['three','four'])

In [18]: df1
Out[18]:
   one  two
a    0    1
b    2    3
c    4    5

In [19]: df2
Out[19]:
   three  four
a      5     6
c      7     8

In [20]: pd.concat([df1,df2],axis=1,keys=['level1','level2'])
Out[20]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

리스트 대신 객체의 사전을 넘기면 사전의 키가 keys 옵션으로 사용된다.

```
In [21]: pd.concat({'level1':df1,'level2':df2},axis = 1)
Out[21]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

계층적 색인을 생성할 때 사용할 수 있는 몇 가지 추가적인 옵션은 아래 표를 참고해라. 새로 생성된 계층의 이름은 names인자로 지정할 수 있다.

```
In [24]: pd.concat([df1,df2],axis = 1,keys=['level1','level2'],
    ...:          names = ['upper','lower'])
Out[24]:
upper level1     level2
lower    one two  three four
a          0   1    5.0  6.0
b          2   3    NaN  NaN
c          4   5    7.0  8.0
```

마지막으로 DataFrame의 로우 색인이 분석에 필요한 데이터를 포함하고 있지 않은 경우 어떻게 할 것인가?

```
In [25]: df1 = pd.DataFrame(np.random.randn(3,4),columns=['a','b','c','d'])

In [26]: df2 = pd.DataFrame(np.random.randn(2,3),columns=['b','d','a'])

In [27]: df1
Out[27]:
          a         b         c         d
0  0.470930 -0.947802  0.114362 -0.068913
1 -0.312762 -0.385400  0.248674  0.325661
2  2.228071 -0.696128 -0.716261  1.367352

In [28]: df2
Out[28]:
          b         d         a
0 -0.307309 -0.347841 -0.984305
1  0.216420 -0.686677 -0.557984
```

이 경우 ignore_index = True 옵션을 주면 된다.

```
In [29]: pd.concat([df1,df2],ignore_index = True )
Out[29]:
          a         b         c         d
0  0.470930 -0.947802  0.114362 -0.068913
1 -0.312762 -0.385400  0.248674  0.325661
2  2.228071 -0.696128 -0.716261  1.367352
3 -0.984305 -0.307309       NaN -0.347841
4 -0.557984  0.216420       NaN -0.686677
```

concat 함수 인자

인자	설명
objs	이어붙일 pandas 객체의 사전이나 리스트, 필수인자
axis	이어붙일 축 방향, 기본값은 0
join	조인 방식, 'inner' (내부 조인, 교집합)과 'outer' (외부 조인, 합집합)가 있으며 기본값은 'outer'
join_axes	합집합/교집합을 수행하는 대신 다른 n-1 축으로 사용할 색인을 지정한다.
keys	이어붙일 객체나 이어붙인 축에 대한 계층 색인을 생성하는 데 연관된 값이다. 리스트나 임의의 값이 들어있는 배열, 튜플의 배열 또는 배열의 리스트가 될 수 있다.
levels	계층 색인 레벨로 사용할 색인을 지정한다. keys가 넘어온 경우 여러 개의 색인을 지정한다.
names	keys나 levels 혹은 둘다 있을 경우 생성된 계층 레벨을 위한 이름
verify_integrity	이어붙인 객체에 중복되는 축이 있는지 검사하고 있다면 예외를 발생시킨다. 기본값은 False로, 중복을 허용한다.
ignore_index	이어붙인 축의 색인을 유지하지 않고 range(total_length)로 새로운 색인을 생성한다.

8.2.4 겹치는 데이터 합치기

데이터를 합칠 때 병합이나 이어붙이기로는 불가능한 상황이 있는데, 두 데이터셋의 색인이 일부 겹치거나 전체가 겹치는 경우가 그렇다. 벡터화된 if-else 구문을 표현하는 NumPy의 where 함수로 자세히 알아보자.

```
In [31]: a = pd.Series([np.nan, 2.5, np.nan,3.5,4.5,np.nan],
    ...:                index = ['f','e','d','c','b','a'])

In [32]: b = pd.Series(np.arange(len(a),dtype=np.float64),
    ...:                index = ['f','e','d','c','b','a'])

In [33]: a
Out[33]:
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64

In [34]: b
Out[34]:
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    5.0
dtype: float64
```

```
In [37]: b[-1]=np.nan

In [38]: b
Out[38]:
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    NaN
dtype: float64

In [39]: np.where(pd.isnull(a),b,a)
Out[39]: array([0. , 2.5, 2. , 3.5, 4.5, nan])
#만약 a가 null 값이면 해당 인덱스의 b 값을 할당함, null 값아니면 그냥 a값 할당
```

Series 객체의 combine_first 메서드는 위와 동일한 연산을 제공하며 데이터 정렬 기능까지 제공한다.

```
In [40]: b[:-2].combine_first(a[2:])
#a의 첫 두개, 나머지는 다 b의 값 재정렬
Out[40]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
dtype: float64
```

DataFrame에서 combine_first 메서드는 컬럼에 대해 같은 동작을 한다. 그러므로 호출하는 객체에서 누락된 데이터를 인자로 넘긴 객체에 있는 값으로 채워 넣을 수 있다.

```
In [45]: df1 = pd.DataFrame({'a':[1.,np.nan,5.,np.nan],
    ...:                     'b':[np.nan, 2., np.nan,6.],
    ...:                     'c':range(2,18,4)})

In [46]: df2 = pd.DataFrame({'a':[5.,4.,np.nan,3.,7.],
    ...:                     'b':[np.nan, 3., 4.,6.,8.]})

In [47]: df1
Out[47]:
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14

In [48]: df2
Out[48]:
     a    b
0  5.0  NaN
1  4.0  3.0
2  NaN  4.0
3  3.0  6.0
4  7.0  8.0
```

```
In [49]: df1.combine_first(df2)
Out[49]:
     a    b     c
0  1.0  NaN   2.0
1  4.0  2.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```

8.3 재형성과 피벗

표 형식의 데이터를 재배치하는 다양한 기본적인 연산이 존재한다. 이런 연산을 **재형성** 또는 **피벗** 연산이라고 한다.

8.3.1 계층적 색인으로 재형성하기

계층적 색인은 DataFrame의 데이터를 재배치하는 다음과 같은 방식을 제공한다.

- **stack**

데이터의 컬럼을 로우로 피벗(또는 회전)시킨다.

- **unstack**

로우를 컬럼으로 피벗시킨다.

몇 가지 예제를 통해 위 연산을 좀 더 알아보자. 문자열이 담긴 배열을 로우와 컬럼의 색인으로 하는 작은 DataFrame이 있다.

```
In [50]: data = pd.DataFrame(np.arange(6).reshape((2,3)),
    ...:                     index = pd.Index(['Ohio','Colorado'],
    ...:                     name='state'),
    ...:                     columns=pd.Index(['one','two','three'],
    ...:                     name='number'))

In [51]: data
Out[51]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

stack 메서드를 사용하면 칼럼이 로우로 피벗되어서 다음과 같은 Series 객체를 반환한다.

```
In [52]: result = data.stack()

In [53]: result
Out[53]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int32
```

unstack 메서드를 사용하면 위 계층적 색인을 가진 Series로부터 다시 DataFrame을 얻을 수 있다.

```
In [54]: result.unstack()
Out[54]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

기본적으로 가장 안쪽에 있는 레벨부터 끄집어내는데(stack도 마찬가지다), 레벨 숫자나 이름을 전달해서 끄집어낼 단계를 지정할 수 있다.

```
In [55]: result.unstack(0)
Out[55]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5

In [58]: result.unstack('state')
Out[58]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

해당 레벨에 있는 모든 값이 하위그룹에 속하지 않을 경우, unstack을 하게 되면 누락된 데이터가 생길 수 있다.

```
In [59]: s1 = pd.Series([0,1,2,3], index=['a','b','c','d'])

In [60]: s2 = pd.Series([4,5,6], index=['c','d','e'])

In [61]: data2 = pd.concat([s1,s2], keys=['one','two'])

In [62]: data2
Out[62]:
one  a    0
     b    1
     c    2
```

```
     d    3
two  c    4
     d    5
     e    6
dtype: int64

In [63]: data2.unstack()
Out[63]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

stack메서드는 누락된 데이터를 자동으로 걸러내기 때문에 연산을 쉽게 원상 복구 할 수 있다.

```
In [64]: data2.unstack().stack()
Out[64]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64

In [65]: data2.unstack().stack(dropna=False)
Out[65]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

DataFrame을 unstack()할 때, unstack 레벨은 결과에서 가장 낮은 단계가 된다.

```
In [66]: df = pd.DataFrame({'left':result,'right':result+5},
    ...:                   columns=pd.Index(['left','right'],name='side'))

In [67]: df
Out[67]:
side             left  right
state    number
Ohio     one        0      5
         two        1      6
         three      2      7
Colorado one        3      8
         two        4      9
         three      5     10

In [69]: df.unstack('state')
```

```
Out[69]:
side   left          right
state  Ohio Colorado  Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

stack을 호출할 때 쌓을 축의 이름을 지정할 수 있다.

```
In [70]: df.unstack('state').stack('side')
Out[70]:
state         Colorado  Ohio
number side
one    left          3     0
       right         8     5
two    left          4     1
       right         9     6
three  left          5     2
       right        10     7
```

8.3.2 긴 형식에서 넓은 형식으로 피벗하기

DB나 CSV 파일에 여러개 시계열 데이터를 저장하는 일반적인 방법은 시간 순서대로 나열하는 것이다. 예제 데이터를 읽어서 시계열 데이터를 다뤄보자.

```
In [74]: data = pd.read_csv('macrodata.csv')

In [75]: data.head()
Out[75]:
     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

[5 rows x 14 columns]

In [76]: periods = pd.PeriodIndex(year = data.year, quarter=data.quarter,
    ...:                          name = 'data')

In [77]: columns = pd.Index(['realgdp','infl','unemp'],name = 'item')

In [78]: data = data.reindex(columns=columns)

In [79]: data.index = periods.to_timestamp('D','end')

In [80]: ldata = data.stack().reset_index().rename(columns={0:'value'})
```

PeriodIndex는 11장에서 제대로 보겠지만, 간단히 설명하면 시간 간격을 나타내기 위한 자료형으로, 연도(year)와 분기(quarter) 컬럼을 합친다.

ldata는 이제 다음과 같다.

```
In [84]: ldata[:10]
Out[84]:
                          data     item     value
0 1959-03-31 23:59:59.999999999  realgdp  2710.349
1 1959-03-31 23:59:59.999999999     infl     0.000
2 1959-03-31 23:59:59.999999999    unemp     5.800
3 1959-06-30 23:59:59.999999999  realgdp  2778.801
4 1959-06-30 23:59:59.999999999     infl     2.340
5 1959-06-30 23:59:59.999999999    unemp     5.100
6 1959-09-30 23:59:59.999999999  realgdp  2775.488
7 1959-09-30 23:59:59.999999999     infl     2.740
8 1959-09-30 23:59:59.999999999    unemp     5.300
9 1959-12-31 23:59:59.999999999  realgdp  2785.204
```

이를 **긴** 형식이라고 부르며, 여러 시계열이나 둘 이상의 키(예제에서는 data와 item)를 가지고 있는 다른 관측 데이터에서 사용한다. 각 로우는 단일 관측치를 나타낸다.

MySQL 같은 관계형 DB는 테이블에 데이터가 추가되거나 삭제되면 item 컬럼에 별개의 값을 넣거나 빼는 방식으로 고정된 스키마(컬럼 이름과 데이터형)에 데이터를 저장한다. 위 예에서 data와 item 은 관계형 DB관점에서 얘기하자면 기본키가 되어 관계 무결성을 제공하며 쉬운 조인 연산과 프로그램에 의한 질의를 가능하게 해준다 물론 단점도 있는데, 길이가 긴 형식으로는 작업이 용이하지 않을 수 있어서 하나의 DataFrame에 data 컬럼의 시간값으로 색인된 개별 item을 컬럼으로 포함시키는 것을 선호할지 모른다.

DataFrame의 pivot 메서드가 바로 이런 변형을 지원한다.

```
In [85]: pivoted = ldata.pivot('data','item','value')

In [86]: pivoted
Out[86]:
item                          infl    realgdp  unemp
data
1959-03-31 23:59:59.999999999  0.00   2710.349    5.8
1959-06-30 23:59:59.999999999  2.34   2778.801    5.1
1959-09-30 23:59:59.999999999  2.74   2775.488    5.3
1959-12-31 23:59:59.999999999  0.27   2785.204    5.6
1960-03-31 23:59:59.999999999  2.31   2847.699    5.2
...                             ...        ...    ...
2008-09-30 23:59:59.999999999 -3.16  13324.600    6.0
2008-12-31 23:59:59.999999999 -8.79  13141.920    6.9
2009-03-31 23:59:59.999999999  0.94  12925.410    8.1
2009-06-30 23:59:59.999999999  3.37  12901.504    9.2
2009-09-30 23:59:59.999999999  3.56  12990.341    9.6

[203 rows x 3 columns]
```

pivot 메서드의 처음 두 인자는 로우와 컬럼 색인으로 사용될 컬럼 이름이고 마지막 두 인자는 DataFrame에 채워 넣을 값을 담고 있는 컬럼 이름이다. 한 번에 두 개의 컬럼을 동시에 변형한다고 하자.

```
In [89]: ldata['value2']=np.random.randn(len(ldata))

In [90]: ldata[:10]
Out[90]:
                          data     item     value    value2
0 1959-03-31 23:59:59.999999999  realgdp  2710.349 -0.409228
1 1959-03-31 23:59:59.999999999     infl     0.000 -1.438547
```

```
2 1959-03-31 23:59:59.999999999    unemp     5.800 -0.909449
3 1959-06-30 23:59:59.999999999  realgdp  2778.801 -1.454837
4 1959-06-30 23:59:59.999999999     infl     2.340 -0.300356
5 1959-06-30 23:59:59.999999999    unemp     5.100  2.678776
6 1959-09-30 23:59:59.999999999  realgdp  2775.488  0.308710
7 1959-09-30 23:59:59.999999999     infl     2.740 -2.074067
8 1959-09-30 23:59:59.999999999    unemp     5.300 -1.118313
9 1959-12-31 23:59:59.999999999  realgdp  2785.204 -0.720559
```

마지막 인자를 생략해서 계층적 컬럼을 가지는 DataFrame을 얻을 수 있다.

```
In [93]: pivoted = ldata.pivot('data','item')

In [94]: pivoted[:5]
Out[94]:
                             value            ...    value2
item                          infl   realgdp  ...   realgdp     unemp
data                                          ...
1959-03-31 23:59:59.999999999  0.00  2710.349  ... -0.409228 -0.909449
1959-06-30 23:59:59.999999999  2.34  2778.801  ... -1.454837  2.678776
1959-09-30 23:59:59.999999999  2.74  2775.488  ...  0.308710 -1.118313
1959-12-31 23:59:59.999999999  0.27  2785.204  ... -0.720559 -0.484585
1960-03-31 23:59:59.999999999  2.31  2847.699  ... -1.530222 -0.869075

[5 rows x 6 columns]

In [95]: pivoted['value'][:5]
Out[95]:
item                           infl   realgdp  unemp
data
1959-03-31 23:59:59.999999999  0.00  2710.349    5.8
1959-06-30 23:59:59.999999999  2.34  2778.801    5.1
1959-09-30 23:59:59.999999999  2.74  2775.488    5.3
1959-12-31 23:59:59.999999999  0.27  2785.204    5.6
1960-03-31 23:59:59.999999999  2.31  2847.699    5.2

In [96]: pivoted['value2'][:5]
Out[96]:
item                              infl   realgdp     unemp
data
1959-03-31 23:59:59.999999999 -1.438547 -0.409228 -0.909449
1959-06-30 23:59:59.999999999 -0.300356 -1.454837  2.678776
1959-09-30 23:59:59.999999999 -2.074067  0.308710 -1.118313
1959-12-31 23:59:59.999999999  1.020965 -0.720559 -0.484585
1960-03-31 23:59:59.999999999  0.288115 -1.530222 -0.869075
```

데이터값으로 사용할 컬럼들의 집합을 지정할 수도 있다.

pivot은 단지 set_index를 사용해서 계층적 색인을 만들고 unstack 메서드를 이용해서 형태를 변경하는 단축키 같은 메서드다.

```
In [100]: unstacked = ldata.set_index(['data','item']).unstack('item')

In [101]: ldata
Out[101]:
```

```
                          data     item      value    value2
0   1959-03-31 23:59:59.999999999  realgdp   2710.349 -0.409228
1   1959-03-31 23:59:59.999999999     infl      0.000 -1.438547
2   1959-03-31 23:59:59.999999999    unemp      5.800 -0.909449
3   1959-06-30 23:59:59.999999999  realgdp   2778.801 -1.454837
4   1959-06-30 23:59:59.999999999     infl      2.340 -0.300356
..                            ...      ...        ...       ...
604 2009-06-30 23:59:59.999999999     infl      3.370 -1.719195
605 2009-06-30 23:59:59.999999999    unemp      9.200 -0.151615
606 2009-09-30 23:59:59.999999999  realgdp  12990.341  1.291612
607 2009-09-30 23:59:59.999999999     infl      3.560 -1.070052
608 2009-09-30 23:59:59.999999999    unemp      9.600 -1.914181

[609 rows x 4 columns]

In [102]: unstacked[:7]
Out[102]:
                          value            ...    value2
item                       infl   realgdp  ...   realgdp     unemp
data                                        ...
1959-03-31 23:59:59.999999999  0.00  2710.349  ... -0.409228 -0.909449
1959-06-30 23:59:59.999999999  2.34  2778.801  ... -1.454837  2.678776
1959-09-30 23:59:59.999999999  2.74  2775.488  ...  0.308710 -1.118313
1959-12-31 23:59:59.999999999  0.27  2785.204  ... -0.720559 -0.484585
1960-03-31 23:59:59.999999999  2.31  2847.699  ... -1.530222 -0.869075
1960-06-30 23:59:59.999999999  0.14  2834.390  ...  0.199877 -1.142591
1960-09-30 23:59:59.999999999  2.70  2839.022  ...  0.626618 -0.285036

[7 rows x 6 columns]
```

8.3.3 넓은 형식에서 긴 형식으로 피벗하기

pivot과 반대되는 연산은 pandas.melt이다. 하나의 컬럼을 여러 개의 새로운 DataFrame으로 생성하기보다는 여러 컬럼을 하나로 병합하고 DataFrame을 입력보다 긴 형태로 만들어낸다.

예제를 보자.

```
In [103]: df = pd.DataFrame({'key':['foo','bar','baz'],
     ...:                    'A':[1,2,3],
     ...:                    'B':[4,5,6],
     ...:                    'C':[7,8,9]})

In [104]: df
Out[104]:
   key  A  B  C
0  foo  1  4  7
1  bar  2  5  8
2  baz  3  6  9
```

'key' 컬럼을 그룹 구분자로 사용할 수 있고 다른 컬럼을 데이터값으로 사용할 수 있다. **pandas.melt를 사용할 때는 반드시 어떤 컬럼을 그룹 구분자로 사용할 것 인지 지정해야 한다.**

여기서는 'key' 를 그룹 구분자로 지정하자.

```
In [105]: melted = pd.melt(df,['key'])

In [106]: melted
```

```
Out[106]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

pivot을 사용해서 원래 모양으로 되돌릴 수 있다.

```
In [107]: reshaped = melted.pivot('key','variable','value')

In [108]: reshaped
Out[108]:
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

pivot의 결과는 로우 라벨로 사용하던 컬럼에서 색인을 생성하므로 reset_index를 이용해서 데이터를 다시 컬럼으로 돌려놓자.

```
In [109]: reshaped.reset_index()
Out[109]:
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

데이터값으로 사용할 컬럼들의 집합을 지정할 수도 있다.

```
In [111]: pd.melt(df,id_vars=['key'],value_vars=['A','B'])
Out[111]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

pandas.melt는 그룹 구분자 없이도 사용할 수 있다.

```
In [112]: pd.melt(df,value_vars = ['A','B','C'])
Out[112]:
  variable  value
0        A      1
1        A      2
2        A      3
3        B      4
```

```
4        B      5
5        B      6
6        C      7
7        C      8
8        C      9

In [113]: pd.melt(df,value_vars = ['key','A','B'])
Out[113]:
  variable value
0      key   foo
1      key   bar
2      key   baz
3        A     1
4        A     2
5        A     3
6        B     4
7        B     5
8        B     6
```

8.3 마치며

지금까지 pandas에서 데이터를 불러오고, 정제하고, 재배열하는 방식을 익혔다. 이제 matplotlib을 이용해 데이터 시각화 단계로 넘어갈 준비가 됐다.