계층적 인덱싱

지금까지 주로 Pandas Series 와 DataFrame 객체에 저장되는 1 차원 2 차원 데이터에 초점을 맞춰 알아봤다. 하지만 종종 한두 개보다 많은 키를 인덱스로 가지는 고차원 데이터를 저장하는 것이 유용할 때가 있다. Pandas 는 기본적으로 3 차원과 4 차원 데이터를 처리할 수 있는 Panel 과 Panel4D 객체를 제공하지만, 실제로 더 일반적으로 사용되는 패턴은 단일 인덱스 내에 여러 인덱스 레벨을 포함하는 계층적 인덱싱(hierarchical indexing, 다중 인덱싱(multi-indexing)이라고도 함)이다. 이 방식으로 고차원 데이터를 익숙한 1 차원 Series 와 2 차원 DataFrame 객체로 간결하게 표현할 수 있다.

이번 절에서는 MultiIndex 객체를 직접 생성하고 다중 인덱스 데이터에서 인덱싱, 슬라이싱, 통계연산을 수행하는 것과 함께 데이터에 대한 단순 인덱스 표현과 계층적 인덱스 표현 간 전환을 위해 사용하는 루틴을 알아보겠다.

```
In [1]: import pandas as pd
        import numpy as np
```

다중 인덱스된 Series

먼저 어떻게 하면 2 차원 데이터를 1 차원 Series 에 표현할 수 있을 지 생각해 보자. 구체적으로 각 점이 문자와 숫자로 이뤄진 키를 갖는 일련의 데이터를 생각해 보자.

나쁜 방식

두 연도에 대해 미국 주의 데이터를 추적한다고 가정해보자. 앞에서 다룬 Pandas 도구를 사용해 간단하게 파이썬 튜플을 키값으로 사용하려고 할 수도 있다.

```
In [2]: index = [('California', 2000), ('California', 2010),
                 ('New York', 2000), ('New York', 2010),
                 ('Texas', 2000), ('Texas', 2010)]
        populations = [33871648, 37253956,
                       18976457, 19378102,
                       20851820, 25145561]
        pop = pd.Series(populations, index=index)
        pop
Out[2]: (California, 2000)    33871648
        (California, 2010)    37253956
        (New York, 2000)      18976457
        (New York, 2010)      19378102
        (Texas, 2000)         20851820
        (Texas, 2010)         25145561
        dtype: int64
```

이 인덱싱 방식을 사용하면 간단하게 이 다중 인덱스를 기반으로 시리즈를 인덱싱하거나 슬라이싱할 수 있다.

```
In [3]:pop[('California', 2010):('Texas', 2000)]
Out[3]:(California, 2010)    37253956
       (New York, 2000)      18976457
       (New York, 2010)      19378102
       (Texas, 2000)         20851820
       dtype: int64
```

그러나 편리함은 거기까지다. 가령 2010 년의 모든 값을 선택해야 한다면 다소 지저분하고 느리기까지 한 데이터 먼징(munging)을 해야 할 것이다.

```
In [4]:pop[[i for i in pop.index if i[1] == 2010]]
Out[4]:(California, 2010)    37253956
       (New York, 2010)      19378102
       (Texas, 2010)         25145561
       dtype: int64
```

이 방식이 원하는 결과를 내주기는 하지만, 지금까지 사용해온 Pandas 의 슬라이싱 구분만큼 깔끔하지도 않고 대규모 데이터의 경우에는 효율적이지도 않다.

더 나은 방식: Pandas MultiIndex

다행히도 Pandas 는 더 나은 방식을 제공한다. 튜플을 기반으로 한 인덱싱은 근본적으로 가장 기초적인 다중 인덱스고, Pandas 의 MultiIndex 타입 이 원하는 유형의 연산을 제공한다. 다음과 같이 튜플로부터 다중 인덱스를 생성할 수 있다.

```
In [5]:index = pd.MultiIndex.from_tuples(index)
       index
Out[5]:MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
                  labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

MultiIndex 는 다중 레벨의 인덱싱을 포함하고 있음을 알아두자. 이 경우에는 주 이름과 연도는 물론이고 이 레벨을 인코딩하는 각 데이터 점에 대한 여러 레이블을 갖고 있다.

이 MultiIndex 를 시리즈로 다시 인덱싱하면 데이터의 계층적 표현을 볼 수 있다.

```
In [6]:pop = pop.reindex(index)
       pop
Out[6]:California  2000    33871648
```

```
              2010    37253956
New York      2000    18976457
              2010    19378102
Texas         2000    20851820
              2010    25145561
dtype: int64
```

여기서 `Series` 표현의 첫 두 열은 다중 인덱스 값을 보여주고, 세번째 열은 그 데이터를 보여준다. 첫 번째 열의 항목 몇 개가 누락돼 있다는 점에 주목하자. 이 다중 인덱스 표현에서 빈 항목은 윗줄과 같은 값을 가리킨다.
이제 두 번째 인덱스가 2010 인 모든 데이터에 접근하려면 간단히 Pandas 슬라이싱 표기법을 사용하면 된다.

```
In [7]:pop[:, 2010]
Out[7]:California    37253956
       New York      19378102
       Texas         25145561
       dtype: int64
```

결과는 관심 있는 키 값 하나로 인덱스가 구성된다. 이 구문은 앞에서 알아본 단순한 튜플 기반의 다중 인덱싱 해법보다 훨씬 더 편리하며 연산도 훨씬 더 효율적이다! 계층적으로 색인된 데이터에 이러한 종류의 인덱싱 연산을 하는 법을 좀 더 알아보자.

MultiIndex : 추가 지원

여기서 아마 다른 점도 눈치챌 수 있을 것이다. 바로 인덱스와 열 레이블을 가진 간단한 `DataFrame` 을 사용해 동일한 데이터를 쉽게 저장할 수 있다는 점이다. 실제로 Pandas 는 이런 유사성을 염두에 두고 만들어졌다. unstack() 메서드는 다중 인덱스를 가진 Series 를 전형적인 인덱스를 가진 DataFrame 으로 빠르게 변환해준다.

```
In [8]:pop_df = pop.unstack()
       pop_df
```

Out[8]:

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

당연히 `stack()` 메서드는 이와 반대되는 연산을 제공한다.

```
In [9]:pop_df.stack()
Out[9]:California  2000    33871648
                   2010    37253956
       New York    2000    18976457
                  2010    19378102
       Texas       2000    20851820
                  2010    25145561
       dtype: int64
```

이것을 보면 계층적 인덱싱을 왜 알아야 하는지 궁금할 것이다. 2 차원 데이터를 1 차원 Series 에 표현하기 위해 다중 인덱싱을 사용할 수 있는 것처럼 3 차원이나 4 차원 데이터를 Series 나 DataFrame 에 표현할 때도 사용할 수 있기 때문이다. 다중 인덱스에서 각 추가 레벨은 데이터의 추가적인 차원을 표현한다. 이 속성을 활용하면 표현할 수 있는 데이터 유형에 훨씬 더 많은 유연성을 제공한다. 구체적으로 연도별 각 주의 인구통계 데이터(예를 들어 18 세 이하 인구수)를 별도의 열로 추가하고 싶을 수도 있다. `MultiIndex` 를 이용하면 이 방식이 `DataFrame` 에 열을 하나추가하는 것만큼 쉽다.

```
In [10]:pop_df = pd.DataFrame({'total': pop,
                               'under18': [9267089, 9284094,
                                           4687374, 4318033,
                                           5906301, 6879014]})
         pop_df
```

Out[10]:

		total	under18
California	**2000**	33871648	9267089
	2010	37253956	9284094
New York	**2000**	18976457	4687374
	2010	19378102	4318033
Texas	**2000**	20851820	5906301
	2010	25145561	6879014

모든 유니버설 함수와 다른 기능들도 계층적 인덱스와 잘 동작한다.
여기서는 위 데이터를 활용해 연도별로 18 세 이하의 인구 비율을 계산한다.

```
In [11]:f_u18 = pop_df['under18'] / pop_df['total']
         f_u18.unstack()
```

Out[11]:

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831

Out[11]:

	2000	2010
Texas	0.283251	0.273568

이렇게 하면 고차원 데이터도 빠르고 쉽게 가공하고 탐색할 수 있다.

MultiIndex 생성 메소드

다중 인덱스를 가진 Series 나 DataFrame 을 생성하는 가장 간단한 방식은 생성자에 2 개 이상의 인덱스 배열 리스트를 전달하는 것이다

```
In [12]: df = pd.DataFrame(np.random.rand(4, 2),
                           index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                           columns=['data1', 'data2'])
         df
```

Out[12]:

		data1	data2
A	1	0.554233	0.356072
	2	0.925244	0.219474
B	1	0.441759	0.610054
	2	0.171495	0.886688

`MultiIndex` 를 생성하는 작업은 백그라운드에서 일어난다.

이와 비슷하게 적당한 튜플을 키로 갖는 딕셔러리를 전달하면 Pandas 는 자동으로 이것을 인식해 기본으로 `MultiIndex` 를 사용한다.

```
In [13]: data = {('California', 2000): 33871648,
                 ('California', 2010): 37253956,
                 ('Texas', 2000): 20851820,
                 ('Texas', 2010): 25145561,
                 ('New York', 2000): 18976457,
                 ('New York', 2010): 19378102}
         pd.Series(data)
Out[13]: California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
```

```
        dtype: int64
```

그렇지만 때로는 명시적으로 `MultiIndex` 를 생성하는 것이 유용할 때가 있다.

명시적 MultiIndex 생성자

인덱스가 생성되는 방법에 더 많은 유연성을 제공하기 위해 `pd.MultiIndex` 의 클래스 메서드 생성자를 사용할 수 있다. 예를 들어, 앞에서 했던 것처럼 각 레벨 내에 인덱스 값을 제공하는 간단한 배열 리스트로 부터 MultiIndex 를 생성할 수 있다.

```
In [14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

그것은 각 점의 여러 인덱스 값을 제공하는 튜플 리스트로부터 생성할 수 있다.

```
In [15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
Out[15]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

심지어 단일 인덱스의 데카르트 곱(Cartesian product)으로부터 MultiIndex 를 생성할 수 있다.

```
In [16]: pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
Out[16]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Series 나 DataFrame 을 생성할 때 index 인수로 이 객체를 기존 Series 나 DataFrame 의 reindex 메서드에 전달할 수 있다.

MultiIndex 레벨 이름

MultiIndex 의 레벨에 이름을 지정하는 것이 편리할 때가 있다. 위의 MultiIndex 생성자에 names 인수를 잔달하거나 생성 후에 인덱스의 name 속성을 설정해 이름을 지정할 수 있다.

```
In [18]: pop.index.names = ['state', 'year']
         pop
Out[18]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

관련 데이터세트가 많으면 이것이 다양한 인덱스 값의 의미를 파악할 수 있는 유용한 방식이 될 수 있다..

열의 MultiIndex

`DataFrame`에서 행과 열은 완전히 대칭적이며 행이 인덱스의 여러 레벨을 가질 수 있듯이 열도 여러 레벨을 가질 수 있다.

```
In [19]: # hierarchical indices and columns 계층적 인덱스와 열
         index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                            names=['year', 'visit'])
         columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                              names=['subject', 'type'])

# mock some data 일부 데이터 모형 만들기
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame 생성하기
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
Out[19]:
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

이 코드를 통해 행과 열 모두에 대한 멀티 인덱싱이 어떤 경우에 매우 유용하게 쓰이는지 알 수 있다. 이것은 기본적으로 4 차원 데이터로, 여기서 차원은 대상(subject), 측정 유형(type), 연도(year), 방문횟수(visit)다. 이것이 있으면 예를 들어 사람 이름으로 최상위 열의 인덱스를 지정하고 그 사람의 정보를 포함하는 전체 DataFrame 을 가져올 수 있다.

```
In [20]: health_data['Guido']
```

Out[20]:

year	visit	type HR	Temp
2013	1	32.0	36.7
	2	50.0	35.0
2014	1	39.0	37.8
	2	48.0	37.3

많은 대상(사람, 국가, 도시 등)에 대해 여러 회에 걸쳐 다중 레이블이 지정된 측정치를 포함하는 복잡한 레코드의 경우, 계층적 행과 열을 사용하는 것이 매우 편리할 수 있다.

MultiIndex 인덱싱 및 슬라이싱

MultiIndex 의 인덱싱과 슬라이싱은 직관적으로 설계했으며, 인덱스를 추가된 차원으로 생각하면 이해하기가 쉽다. 먼저 다중 인덱스를 가진 Series 인덱싱을 살펴본 다음, 다중 인덱스를 가진 DataFrame 을 살펴본다.

다중 인덱스를 가진 Series

앞에서 본 다중 인덱스를 가진 주 별 인구수 Series 를 생각해 보자.

```
In [21]:pop
Out[21]:state       year
        California  2000    33871648
                    2010    37253956
        New York    2000    18976457
                    2010    19378102
        Texas       2000    20851820
                    2010    25145561
        dtype: int64
```

여러 용도로 인덱싱해서 단일 요소에 접근할 수 있다

```
In [22]:pop['California', 2000]
Out[22]:33871648
```

MultiIndex 는 부분인덱싱(parial indexing)이나 인덱스 레벨 중 하나만 인덱싱하는 것도 지원한다. 그 결과 더 낮은 수준의 인덱스를 유지하는 다른 Series 를 얻게 된다.

```
In [23]:pop['California']
Out[23]:year
        2000    33871648
```

```
        2010    37253956
        dtype: int64
```

MultiIndex 가 정렬돼 있다면 부분 슬라이싱도 가능하다.

```
In [24]:pop.loc['California':'New York']
Out[24]: state       year
California  2000    33871648
           2010    37253956
New York    2000    18976457
           2010    19378102
dtype: int64
```

인덱스가 정렬돼 있다면 첫 번째 인덱스에 빈 슬라이스를 전달함으로써 더 낮은 레벨에서 부분 인덱싱을 수행 할 수 있다.:

```
In [25]:pop[:, 2000]
Out[25]:state
        California    33871648
        New York      18976457
        Texas         20851820
        dtype: int64
```

다른 유형의 데이터 인덱싱과 선택 방식 역시 적용할 수 있다. 예를 들면 부울 마스크를 이용해 데이터를 선택할 수 있다.:

```
In [26]:pop[pop > 22000000]
Out[26]:state       year
        California  2000    33871648
                    2010    37253956
        Texas       2010    25145561
        dtype: int64
```

팬시 인덱싱(fancy Indexing)을 이용한 데이터 선택도 가능하다.

```
In [27]:pop[['California', 'Texas']]
Out[27]:state       year
        California  2000    33871648
                    2010    37253956
         Texas      2000    20851820
                    2010    25145561
        dtype: int64
```

다중 인덱스를 가진 DataFrames

다중 인덱스를 가진 DataFrame 도 비슷한 방식으로 동작한다. 앞에서 만든 의료 DataFrame 을 생각해 보자.

```
In [28]: health_data
```

Out[28]:

	subject	Bob		Guido		Sue	
	Type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

열은 DataFrame 의 기본 요소이며, 다중 인덱스를 가진 Series 에서 사용된 구문이 열에 적용된다는 사실을 기억하라. 예를 들어, Guido 의 심장박동 수 데이터를 간단한 연산으로 가져올 수 있다.

```
In [29]: health_data['Guido', 'HR']
Out[29]: year  visit
         2013  1       32.0
               2       50.0
         2014  1       39.0
               2       48.0
         Name: (Guido, HR), dtype: float64
```

또한, 단일 인덱스의 경우의 마찬가지로 loc, iloc, ix 인덱스를 사용할 수도 있다.

```
In [30]: health_data.iloc[:2, :2]
```

Out[30]:

	subject	Bob	
	type	HR	Temp
Year	visit		
2013	1	31.0	38.7
	2	44.0	37.7

이 인덱서는 기반이 되는 2 차원 데이터를 배열처럼 보여주지만, loc 나 iloc 에서 갭별 인덱스는 다중 인덱스의 튜플로 전달될 수 있다.

```
In [31]: health_data.loc[:, ('Bob', 'HR')]
Out[31]: year  visit
         2013  1       31.0
               2       44.0
         2014  1       30.0
               2       47.0
         Name: (Bob, HR), dtype: float64
```

이 인덱스 튜플 내에서 슬라이스로 작업하는 것은 그다지 편리하지 않다. 튜플 내에 슬라이스를 생성하려고 하면 구문 에러가 발생할 것이다.

```
In [32]: health_data.loc[(:, 1), (:, 'HR')]
  File "<ipython-input-32-8e3cc151e316>", line 1
    health_data.loc[(:, 1), (:, 'HR')]
                    ^
SyntaxError: invalid syntax
```

파이썬 기본 함수인 slice()를 사용해 원하는 슬라이스를 명시적으로 만들면 이러한 에러를 피할 수 있지만 Pandas 가 정확히 이러한 상황을 고려해 제공하는 IndexSlice 객체를 사용하는 것이 더 낫다.

```
In [33]: idx = pd.IndexSlice
         health_data.loc[idx[:, 1], idx[:, 'HR']]
Out[33]:
```

	subject	Bob	Guido	Sue
	type	HR	HR	HR
year	visit			
2013	1	31.0	32.0	35.0
2014	1	30.0	39.0	61.0

다중 인덱스를 가진 Series 와 DataFrame 의 데이터와 상호작용하는 방법은 많이 있으며 그것들을 실제로 사용해 보면서 익숙해지는 것이 가장 좋다.

다중 인덱스 재정렬하기

다중 인덱스를 가진 데이터를 사용할 때 가장 중요한 점의 하나는 데이터를 효과적으로 변환하는 방법을 아는 것이다. 데이터세트의 모든 정보를 보존하지만 다양한 연산의 목적에 따라 그 정보를 재정렬하는

연산이 많이 있다. 이에 대한 간단한 예제로 이미 stack()과 unstack()메서드를 살펴봤지만 그것 말고도 계층적 인덱스와 열 사이에서 데이터를 재 정렬하는 방식을 세밀하게 조정할 수 있는 방법은 많이 있다.

정렬된 인덱스와 정렬되지 않은 인덱스

대부분의 MultiIndex 슬라이싱 연산은 인덱스가 정렬돼 있지 않으면 실패한다.

우선 인덱스가 사전적으로 정렬돼 있지 않은 다중 인덱스를 갖는 간단한 데이터를 만들어보자

```
In [34]: index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
         data = pd.Series(np.random.rand(6), index=index)
         data.index.names = ['char', 'int']
         data
Out[34]: char  int
         a     1      0.003001
               2      0.164974
         c     1      0.741650
               2      0.569264
         b     1      0.001693
               2      0.526226
         dtype: float64
```

이 인덱스를 부분 슬라이싱하려고 하면 오류가 발생한다.

```
In [35]: try:
             data['a':'b']
         except KeyError as e:
             print(type(e))
             print(e)
         <class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

에러 메시지에서 명확하게 드러나지는 않지만, 이것은 MultiIndex 가 정렬되지 않아서 나타나는 결과다. 여러가지 이유로 부분 슬라이스와 그와 유사한 다른 연산을 수행하려면 MultiIndex 의 레벨이 정렬된(즉, 사전적)순서를 가져야 한다. Pandas 는 이러한 유형의 정렬을 수행하는 다수의 편리한 루틴을 제공한다. DataFrame 의 sort_index()와 sortlevel()메서드를 예로 들 수 있다.

```
In [36]: data = data.sort_index()
         data
Out[36]: char  int
         a     1      0.003001
               2      0.164974
         b     1      0.001693
               2      0.526226
         c     1      0.741650
               2      0.569264
         dtype: float64
```

이 방식으로 정렬된 인덱스를 사용하면 부분 슬라이싱은 예상대로 동작한다.

```
In [37]:data['a':'b']
Out[37]:char  int
        a     1      0.003001
              2      0.164974
        b     1      0.001693
              2      0.526226
        dtype: float64
```

인덱스 스태킹 및 언스태킹

데이터를 정렬된 다중 인덱스에서 간단한 2 차원 표현으로 변경할 수 있으며, 이때 선택적으로 사용할 레벨을 지정할 수 있다.

```
In [38]:pop.unstack(level=0)
Out[38]:
```

state	California	New York	Texas
year			
2000	33871648	18976457	20851820
2010	37253956	19378102	25145561

```
In [39]:pop.unstack(level=1)
Out[39]:
```

Year	2000	2010
State		
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

unstack() 의 역은 stack()으로, 원래 시리즈로 회복하는데 사용할 수 있다.

```
In [40]:pop.unstack().stack()
Out[40]:
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
```

```
          2010    25145561
dtype: int64
```

인덱스 설정 및 재설정

계층적 데이터를 재정렬하는 또 다른 방법은 인덱스 레이블을 열로 바꾸는 것으로, reset_index 메서드 수행할 수 있다. 인구 딕셔너리 pop 에서 이 메서드를 호출하면 전에 인덱스에 있던 정보를 그대로 유지하는 state 와 year 열을 가진 DataFrame 을 얻게 된다. 명확성을 위해 선택적으로 열에 표현할 데이터의 이름을 지정할 수 있다.

```
In [41]: pop_flat = pop.reset_index(name='population')
         pop_flat
Out[41]:
```

	State	year	population
0	California	2000	33871648
1	California	2010	37253956
2	New York	2000	18976457
3	New York	2010	19378102
4	Texas	2000	20851820
5	Texas	2010	25145561

종종 실제 데이터로 작업하다 보면 위와 같은 원시 입력 데이터를 만나기도 하는데, 이때 열 값으로부터 MultiIndex 를 만드는 것이 유용하다. 이 작업은 다중 인덱스를 갖는 DataFrame 을 반환하는 DataFrame 의 set_index 메서드로 할 수 있다.

```
In [42]: pop_flat.set_index(['state', 'year'])
Out[42]:
```

		population
state	year	
California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102
Texas	2000	20851820

		population
state	year	
	2010	25145561

실제로 이러한 유형의 인덱스 재정렬 작업은 현실 세계의 데이터세트를 만났을 때 매우 유용한 활용 패턴의 하나다.

다중 인덱스에서 데이터 집계

앞에서 Pandas 가 기본적으로 mean (), sum(), max()와 같은 데이터 집계 메서드를 제공하는 것을 봤다. 계층적 인덱스를 가진 데이터의 경우, 데이터의 어느 부분 집합에 대해 집계 연산을 수행할 것인지 제어하는 level 매개변수를 집계 메서드에 전달할 수 있다.

앞에서 다른 의료 데이터로 돌아가 보자.

```
In [43]: health_data
Out[43]:
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

해마다 두 번의 방문에서 얻은 측정치의 평균을 구하려고 한다. 이 작업은 탐색하고자 하는 인덱스 레벨의 이름을 지정해서 할 수 있는데, 이 경우에는 연도다.

```
In [44]: data_mean = health_data.mean(level='year')
         data_mean
Out[44]:
```

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	37.5	38.2	41.0	35.85	32.0	36.95

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2014	38.5	37.6	43.5	37.55	56.0	36.70

더 나아가 axis 키워드를 사용해 열의 레벨 간 평균을 취할 수도 있다.

```
In [45]: data_mean.mean(axis=1, level='type')
Out[45]:
```

type	HR	Temp
year		
2013	36.833333	37.000000
2014	46.000000	37.283333

이처럼 코드 두 줄로 모든 대상이 매년 모든 방문에서 측정한 평균 심박 수와 체온을 알아낼 수 있었다.

Aside: Panel Data

Pandas 에는 아직 논의하지 않은 몇가지 다른 기본 자료구조가 있다. Pd.Panel4D 객체가 여기에 해당한다. 이것들은 각각 1 차원 Series 와 2 차원 DataFrame 구조를 3 차원과 4 차원으로 일반화한 구조라고 생각하면된다. Series 와 DataFrame 의 인덱싱과 각공 방식에 믹숙해지면 Panel 과 Panel4D 는 비교적 사용하기 쉽다. Ix,loc, iloc 인덱서는 이 고차원 구조로 쉽게 확장될 수 있다.

패널 데이터는 조밀한 데이터(dense data)를 표현하지만 다중 인덱싱은 희박한 데이터(sparse data)를 표현한다. 차원 수가 증가함에 따라 조밀한 표현방식은 대부분의 현실 세계데이터세트에는 매우 비효율적일 수 있다. 하지만 간혹 특수한 애플리케이션에서 이 구조가 유용할 수도 있다.